

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



Vue2

实践揭秘

梁睿坤 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



作者简介

十余年软件开发、项目管理、团队建设经验。长年致力于互联网技术应用以及大数据应用方面的研究与开发工作。曾任多家软件公司的高级软件工程师、项目经理、首席架构师、技术总监等职。

曾任广州市优晟网络股份有限公司技术总监，从事微信开发以及大数据在电商与互联网传播方面的应用。目前主要从事软件工程、系统架构、语言基础以及 IoT、大数据与 AI 在商业应用方面的研究与实践。

联系方式

博客园: <http://www.cnblogs.com/ray-liang>

简书: <http://www.jianshu.com/u/5c81d3d72b56>

邮箱: csharp2002@hotmail.com

微博: 广州亚睿

GitHub: <https://github.com/dotnetage>



Vue2

实践揭秘

—— 梁睿坤 著 ——

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书以 Vue2 的实践应用为根基,从实际示例入手,详细讲解 Vue2 的基础理论应用及高级组件开发,通过简明易懂的实例代码,生动地让读者快速、全方位地掌握 Vue2 的各种入门技巧以及一些在实际项目中的宝贵经验。

本书除了全面、细致地讲述 Vue2 的生态结构、实际编程技巧和一些从实践中得到的经验,还重点介绍如何以组件化编程思想为指导,以前端工程化方法为实现手段来实践 Vue2,通过组件的单元测试和 E2E 测试来保证工程质量。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Vue2 实践揭秘 / 梁睿坤著. —北京:电子工业出版社, 2017.4
ISBN 978-7-121-31068-3

I. ①V… II. ①梁… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 047565 号

责任编辑:陈晓猛

印 刷:三河市双峰印刷装订有限公司

装 订:三河市双峰印刷装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:19 字数:400 千字

版 次:2017 年 4 月第 1 版

印 次:2017 年 4 月第 1 次印刷

定 价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819, faq@phei.com.cn。

前言

写作背景

我从事软件开发至今接近 18 年了，在我进入这个行业之前，只有软件工程师与硬件工程师之分，并没有什么前端工程师或者后端工程师之说。前端后端都得懂，否则根本找不到工作。当然，现在对于前端工程师与后端工程师的划分是由于软件技术发展实在太快了，两个方向已经发展成各自独立的体系，前端开发由于技术的革新、移动端的崛起，其地位显得越来越重要。

我是从 jQuery 1.0 开始真正地接受前端工程化开发概念的，也是从那时对 JavaScript 产生了一发而不可收的兴趣。因为前端工程化能使项目的体系结构更加合理，那些在后端实现起来极为繁杂的交互操作以一种最“轻巧”的方式给取代了。当第一次遇到 Angular 之时我更是兴奋不已，它简直就是为传统工程师或者说是后端工程师配备的最强大的前端武器！虽然 Angular 的入门曲线非常陡峭，很多内容都极为晦涩难懂，但它与 jQuery 一样，可以算得上是前端架构发展史上的一座丰碑。

软件领域中后者永远具有更大的吸引力，在前端开发领域，React 可以说是继 Angular 之后又一震撼整个前端开发圈子的重磅炸弹。与 Angular 相比，它大大降低了学习的成本，同时拥有极高效的运行效能，使之一下子盖过了 Angular 的风头。Angular 与 React 两套前端框架的崛起也掀起了整个前端开发圈的一股革命，实际上我们都清楚这是 Google 与 Facebook 之间对开发者的一种争夺手段。对于一直从事实战领域应用的开发者而言，虽然有更多的选择是好事，但“谁更好用？”，“谁更强大？”这类选择困难症也将伴随而来。

Angular 与 React 各有优劣，很难从综合性上来评判谁比谁更好，加上 Angular2 的诞生，使得我们更难以从中选择最合心意的框架了，可能最熟悉的就自然成为最好的了吧。

2016 年我和我的团队所从事的 Web 项目由于需要有大量界面交互功能，因此我引入了 Angular2 + Flask 的搭配方式作为项目的基础语言架构。然而，我的团队大多数是由从事多年后端开发和系统开发的工程师所组成，他们对当下前沿的前端技术涉猎并不算深入，我只能不断地进行内部培训以快速提升团队的前端开发实力。Angular2 一直处于 Beta 状态，

而且相关的官方开发文档一直缺失，开发与测试工具的发展也相对滞后，在实际使用过程中，TypeScript 这个将弱类型化的 JS 强制变成强类型语言的怪胎在不断地给我们制造麻烦，除了让团队接受 Angular2 对 Angular 的优化模式，还得不断地在各种大坑中求生存，这毫无疑问对于我和我的团队是一种极大的挑战。当时我非常担心由于选择了 Angular2 而导致项目失败，中途曾想过用 React 对之加以取代。但从实际出发，这只是一种换汤不换药的方案而已，直至我们偶然间遇到了 Vue，Vue 可以说给予我们项目生的希望！选中了 Vue 是因为我和我的团队只是付出了极小的代价，甚至可以说是毫无障碍地将 Angular2 上开发的代码切换到 Vue 上面来，Vue 的开发工具链虽说没有 Angular.js 完备，但有 vue-cli 的辅助，也基本能应付项目开发的需要，架构理论上几乎就是对 Angular.js 的简化。更吸引我们的是，这是一个由我们中国人开发的前端框架！而且适合我们项目使用的社区资源也非常丰富，性能、工具链、学习曲线、极小的运行库这些优点一下就完全弥补了 Angular 的不足，也成为了我们项目最后能守住的最坚实的防线。

编撰此书出于一次巧合，我们在升级到 Vue2 之后我一直想找一本能系统化、全面地讲述 Vue2 开发的书籍作为我团队的培训教材，但很可惜一直无法找到。出于一时的心血来潮，突然间想将我们在实践中应用 Vue2 的一些技巧和方法记录下来编撰成书，此时也得到了本书的策划编辑陈晓猛先生给予我的鼓励与支持才得以成书。

此书从构思到成书用了接近 4 个月，实际上花在编撰上的时间估计也只是一个月左右，其他的时间都用在在了准备素材与写代码上。本书中的素材都取自我参与过的项目，在此过程中我对 Vue2 的实践应用也有了很大的提高与深化。期望此书能为正在奋斗于前端开发工作的同行们带来帮助，同时也作为我对 Vue 团队的一种支持。Vue 是一款能与世界级的 Angular 与 React 比肩的前端框架，更重要的是它是由我们中国人“智造”的！

内容介绍

本书以 Vue2 的理论为中心，以实战示例为基础，通过示例应用展开覆盖 Vue 的各个理论知识点。本书从实践应用出发，对 Vue 官方未曾进行详尽说明甚至不曾提及的实用内容进行揭秘，试图使此书能成为你在 Vue 前端工程化开发实战中的参考手册。本书主要从多个示例由浅入深地讲述 Vue 的使用知识，除此之外，还重点介绍了 Vue 工程化开发中必备的源码库、第三方开发工具以及如何对 Vue 的各种模块进行全方位的测试。

第 1 章 从一个经典的“待办事项”(TODOs) 示例入手，从零开始介绍 Vue 的入门知识，包括插值、数据绑定、属性与样式绑定和组件的基本概念与用法。

第2章 讲述如何为 Vue 建立一个真实的工程化开发的环境，以及工程化环境下第三方工具的基本使用与配置，其中包括：vue-cli、webpack、Karma、Phantom、Mocha、Sinon、Chai 和 Nightwatch。

第3章 介绍 Vue 的路由机制和 Vue 生态系统中最重要的一员——vue-router 的基本使用方法。

第4章 通过手机书店示例来介绍组件化理论与 Vue 组件的设计与实现的具体方法，包括抽象组件的基本方法，如何用 Vue 对组件进行封装，如何从界面中提取公共的数据接口，如何在没有实现服务端的情况下运行 Vue 程序以及怎样创建复杂的复合型组件。

第5章 全方位地讲述 Vue 的测试与调试过程中使用到的技术与工具，包括 Mocha 的使用方法，如何为组件编写单元测试，如何在运行期和单元测试中进行调试，如何进行端对端测试。

第6章 通过一个非常普遍且实用的图书管理示例讲述 Vue 在实现一个具有复杂操作的界面时所采用的技术知识点，以及 Vue 组件的高级用法。例如视图的排序、分页、查找，多行删除的设计与实现，通过表单处理图书数据的添加、编辑和数据验证，如何用组件化的设计方法封装 Vue 组件以实现最大限度的组件重用。

第7章 介绍 Vue 生态结构中针对规模庞大的前端程序进行状态管理的利器 Vuex，通过实例对 Vuex 的应用原则和结构组成进行一一剖析，讲述如何将各种本来混乱的组件状态通过 Vuex 来将其进行分离，每个部分应该如何设计与编码，如何进行测试，最终使 Vue 前端工程架构变得更为合理。

本书相关源码

- 本书源码汇总——<https://github.com/DotNetAge/vue-in-action>;
- vue-todos——第1章例说 Vue.js 的示例源码 <https://github.com/DotNetAge/vue-todos>;
- vue-shop——第3章路由与页面间导航的示例源码 <https://github.com/DotNetAge/vue-shop>;
- vue-curd——第6章视图与表单的处理的示例源码 <https://github.com/DotNetAge/vue-curd>;
- V-UIKit——UIKit for Vue2 的组件库，构思源于第4章组件化的设计与实现方法的内

容 <https://dotnetage.github.io/vue-ui/>;

- vue-nvd3——基于 NVD3 开发的 Vue2 的组件库 <https://github.com/DotNetAge/vue-nvd3>;
- vue-easy-pie-chart——基于 easy pie chart 开发的环状图组件库 <https://github.com/DotNetAge/vue-easy-pie-chart>。

勘误和交流

本书如有勘误，会在 <https://github.com/DotNetAge/vue-in-action> 上发布。由于笔者能力有限，时间仓促，书中难免有错漏，欢迎读者批评指正。读者也可以到博文视点官网的本书页面进行交流（www.broadview.com.cn/31068）。注册成为博文视点社区用户，可享受以下服务：

- 下载资源：本书所提供的示例代码及资源文件均可在【下载资源】处下载。
- 提交勘误：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 与我交流：在页面下方【读者评论】处留下您的疑问或观点，与我和其他读者一同交流。
- 页面入口：



您也可以直接联系我：

- 博客园：<http://www.cnblogs.com/ray-liang>
- 简书：<http://www.jianshu.com/u/5c81d3d72b56>
- 邮箱：csharp2002@hotmail.com
- 微博：广州亚睿
- GitHub：<https://github.com/dotnetage>

致谢

首先，感谢电子工业出版社博文视点公司的陈晓猛编辑，是您鼓励我将本书付诸成册，并在我写作过程中审阅了大量稿件，给予我很多指导和帮助。感谢工作在幕后的电子工业出版社评审团队对于本书在校对、排版、审核、封面设计、错误改进方面所给予的帮助，使本书得以顺利出版发行。其次，感谢在我十几年求学生涯中教育过我的所有老师，是你们将知识和学习方法传递给了我。感谢我曾经工作过的公司和单位，感谢和我一起共事过的同事和战友，你们的优秀一直是我追逐的目标，你们所给予的压力正是我不断改进自己的动力。

感谢我的父母和儿子。由于撰写本书，牺牲了很多陪伴家人的时间。感谢你们对我工作的理解和支持。

2017年2月16日梁睿坤于广州

目 录

第 1 章 例说 Vue.js	1
1.1 插值	5
1.2 数据绑定	6
1.3 样式绑定	9
1.4 过滤器	12
第 2 章 工程化的 Vue.js 开发	15
2.1 脚手架 vue-cli	16
2.2 深入 vue-cli 的工程模板	19
2.2.1 webpack-simple 模板	19
2.2.2 webpack 模板	21
2.2.3 构建工具	23
2.3 Vue 工程的 webpack 配置与基本用法	25
2.3.1 webpack 的特点	26
2.3.2 基本用法	27
2.3.3 用别名取代路径引用	29
2.3.4 配置多入口程序	30
2.4 基于 Karma+Phantom+Mocha+Sinon+Chai 的单元测试环境	32
2.5 基于 Nightwatch 的端到端测试环境	38
第 3 章 路由与页面间导航	51
3.1 vue-router	53
3.2 路由的模式	57
3.3 路由与导航	58
3.4 导航状态样式	69
3.5 History 的控制	70
3.6 关于 Fallback	71

3.7 小结	73
第4章 页面的区块化与组件的封装	75
4.1 页面逻辑的实现	76
4.2 封装可重用组件	80
4.3 自定义事件	87
4.4 数据接口的分析与提取	89
4.5 从服务端获取数据	91
4.6 创建复合型的模板组件	95
4.7 数据模拟	100
4.8 小结	102
4.9 扩展阅读: Vue 组件的继承——mixin	103
第5章 Vue 的测试与调试技术	110
5.1 Mocha 入门	111
5.2 组件的单元测试方法	118
5.3 单元测试中的仿真技术	121
5.3.1 调用侦测 (Spies)	124
5.3.2 Sinon 的断言扩展	126
5.3.3 存根 (stub)	128
5.3.4 接口仿真 (Mocks)	131
5.3.5 后端服务仿真	133
5.4 调试	134
5.5 Nightwatch 入门	139
5.5.1 编写端到端测试	139
5.5.2 钩子函数与异步测试	141
5.5.3 全局模块与 Nightwatch 的调试	143
5.5.4 Page Objects 模式	147
第6章 视图与表单的处理	153
6.1 为 Vue2 集成 UIKit	154
6.2 表格视图的实现	159
6.2.1 实时数据筛选	164
6.2.2 多行数据的选择	167

6.2.3 排序的实现	171
6.3 单一职责原则与高级组件开发方法	176
6.3.1 搜索区的组件化	177
6.3.2 母板组件	179
6.3.3 重构模态对话框组件	181
6.3.4 高级组件与 Render 方法	183
6.3.5 UIKit 按钮	194
6.3.6 通用表格组件	198
6.4 表单的设计与实现	211
6.4.1 计算属性的双向绑定	214
6.4.2 富文本编辑器组件的实现	215
6.4.3 实现嵌套式容器组件	220
6.4.4 表单的验证	224
6.5 集成服务端的 CRUD Restful API	239
6.6 HTTP 拦截器 inteceptor	242
6.7 开发服务器的定制	245
第 7 章 Vuex 状态管理	250
7.1 Vuex 的基本结构	253
7.2 data 的替代者——State 和 Getter	256
7.3 测试 Getter	260
7.4 Action——操作的执行者	261
7.5 测试 Action	263
7.6 只用 Mutation 修改状态	265
7.7 测试 Mutations	268
7.8 子状态和模块	269
7.9 用服务分离外部操作	274
附录 A Chai 断言参考	277
附录 B Vee-Validate 验证规则参考	289

第 1 章 例说 Vue.js

本章将通过极具代表性的 Todo 的示例作为引领读者进入 Vue.js 大门的引子。我会以实践为第一出发点，从零开始一步一步地构造一个单页式的 Todo 应用，在这个过程中会将 Vue.js 相关的知识点融入其中，在实际应用中展现这个“小”而“强”的界面框架。

我们先来看看最终希望构造出一个什么样的 App:



Vue.js 与 Angular2 和 React 相比，让我感觉最舒适的是它在一开始就为我们铺平了入门的道路，这就是它的脚手架 vue-cli。因为它的存在，省去了手工配置开发环境、运行环境和测试环境的步骤，开发者可以直接步入 Vue.js 开发的殿堂。然而，现在我并不打算详细地介绍这个脚手架工具，先让我们一起从使用体验来感性认识它，在后面的章节中我会详细地介绍这个工具。

在开始动手之前，必须先得在机器上安装好 npm，然后输入以下指令将 vue-cli 安装到机器的全局环境中：

```
$ npm i vue-cli -g
```

然后，我们就可以开始建立工程了，键入以下的指令：

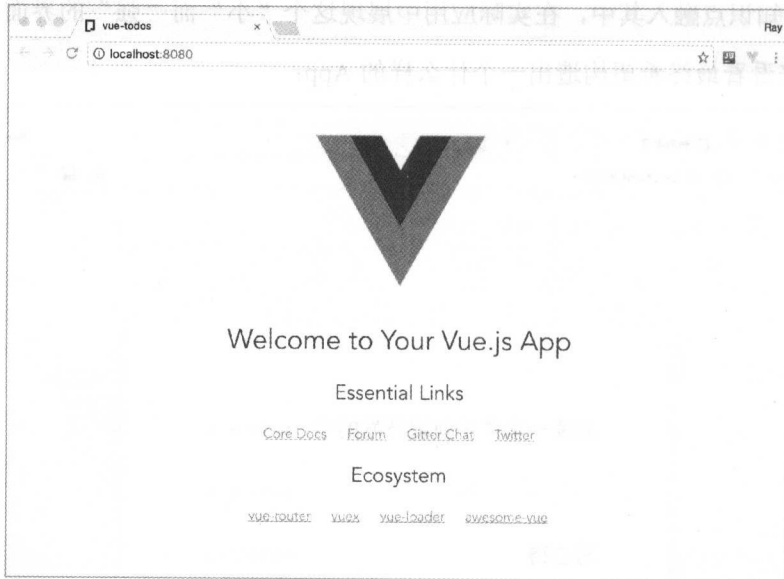
```
$ vue init webpack-simple vue-todos
```

此时控制台会提出一些关于这个新建项目的基本问题，直接“回车”跳过就行了。然后进入 `vue-todo` 目录，安装脚手架项目的基本支持包：

```
$ npm i
```

安装完支持包后键入以下指令就可以运行一个由脚手架构建的基本 Vue.js 程序了：

```
$ npm run dev
```



是不是很简单？进入代码中看看 `vue-cli` 到底为我们构造了一个什么样的代码结构：

```
├── README.md
├── index.html      # 默认启动页面
├── package.json   # npm 包配置文件
├── src
│   ├── App.vue    # 启动组件
│   ├── assets
│   │   └── logo.png
│   └── main.js     # Vue 实例启动入口
└── webpack.config.js # webpack 配置文件
```

Vue2 与 Vue1.x 相比有了很大的区别，从最小化的运行程序开始了解 Vue 是一种绝佳的途径，先从 `main.js` 文件入手：

```
import Vue from 'vue'
import App from './App.vue'

new Vue({
  el: '#app',
  render: h => h(App)
})
```

这里就运用了 Vue2 新增的特色 **Render** 方法，如果你曾用过 **React**，是不是有一种似曾相识之感？确实，Vue2 甚至连渲染机制都与 **React** 一样了。为了得到更好的运行速度，Vue2 也采用了 **Virtual DOM**。如果你还没有接触过 **Virtual DOM**，并不要紧，现在只需要知道它是一种比浏览器原生的 **DOM** 具有更好性能的虚拟组件模型就行了，我们会在稍后的章节中再来讨论它。

我们需要知道的是，通过 **import** 将一个 **Vue.js** 的组件文件引入，并创建一个 **Vue** 对象的实例，在 **Vue** 实例中用 **Render** 方法来绘制这个 **Vue** 组件（**App**）就完成了初始化。

然后，将 **Vue** 实例绑定到一个页面上，真实存在的元素 **App** **Vue** 程序就引导成功了。

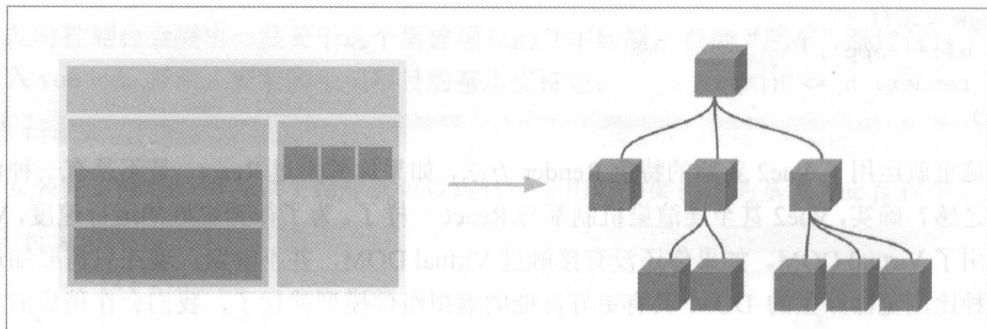
打开 **index.html** 文件就能看到 **Vue** 实例与页面的对应关系：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <!-- Vue 实例所对应的页面元素 -->
  <div id="app"></div>
  <!-- 由 Webpack 编译后的运行文件 -->
  <script src="/dist/build.js"></script>
</body>
</html>
```

也就是说，一个 **Vue** 实例必须与一个页面元素绑定。**Vue** 实例一般用作 **Vue** 的全局配置来使用，例如向实例安装路由、资源插件，配置应用于全局的自定义过滤器、自定义指令等。在本章示例中，我们只需要知道它的作用就可以了。

我们需要了解的是 **App.vue** 这个文件，***.vue** 是 **Vue.js** 特有的文件格式，表示的就是一个 **Vue** 组件，它也是 **Vue.js** 的最大特色，被称为单页式组件。“***.vue**”文件可以同时承载“视图模板”、“样式定义”和组件代码，它使得组件的文件组织更加清晰与统一。

Vue.js 的组件系统提供了一种抽象，让我们可以用独立可复用的小组件来构建大型应用。如果我们考虑到这一点，几乎任意类型应用的界面都可以抽象为一个组件树：



Vue2 具有很高的兼容性，我们也可以用“js”文件来单纯地定义组件的逻辑，甚至可以使用 React 的 JSX 格式的组件（需要 babel-plugin-transform-vue-jsx 支持）。

脚手架为我们创建的这个 App 组件内加入了不少介绍性的文字，将这个文件“净化”后就可以得到一个最简单的 Vue 组件定义模板：

```
<template>
  <div id="app">
  </div>
</template>

<style></style>

<script>
export default {
  name: 'app'
}
</script>
```

由以上的代码我们可以了解到，单页组件由以下三个部分组成：

- <template>——视图模板；
- <style>——组件样式表；
- <script>——组件定义。

接下来我们就从这个示例开始，一步步学习 Vue 的基本组成部分，在实践中理解它们的作用。

1.1 插值

Vue 的视图模板是基于 DOM 实现的。这意味着所有的 Vue 模板都是可解析的有效的 HTML，而且它对一些特殊的特性做了增强。接下来，我们就在模板上定义一个网页标题，并通过数据绑定语法将 App 组件上定义的数据模型绑定到模板上。

首先，在组件脚本定义中使用 `data` 定义用于内部访问的数据模型：

```
export default {
  ...
  data () {
    return {
      title: "vue-todos"
    }
  }
}
```

`data` 可以是一个返回 Object 对象的函数，也可以是一个对象属性，也就是说，可以写成以下方式：

```
export default {
  ...
  data : {
    title: "vue-todos"
  }
}
```

使用函数返回是为了可以具有更高的灵活性，例如对内部数据进行一些初始化的处理，官方推荐的用法是采用返回 Object 对象的函数。

在模板中引用 `data.title` 数据时我们并不需要写上 `data`，这只是 Vue 定义时的一个内部数据容器，通过 Vue 模块的插值方式直接写上 `title` 即可：

```
<h1>{{ title }}</h1>
```

用双大括号 `{{ }}` 引住的内容被称为“Mustache”语法，Mustache 标签会被相应数据对象的 `title` 属性的值替换。每当这个属性变化时它也会更新。

插值是 Vue 模板语言的最基础用法，很多的变量输出都会采用插值的方式，而且插值还可以支持 JavaScript 表达式运算和过滤器（下文将会提及）。`{{ }}` 引用的内容都会被编码，如果要输出未被编码的文本，可以使用 `{{{ }}` 对变量进行引用。

完整代码如下所示。

```
<template>
  <div id="app">
    <h1>{{ title }}</h1>
  </div>
</template>

<style></style>

<script>
export default {
  name: 'app',
  data () {
    return {
      title: "vue-todos"
    }
  }
}
</script>
```

从 Vue2 开始，组件模板必须且只能有一个顶层元素，如果在组件模块内设置多个顶层元素将会引发编译异常。

请注意，在上述代码中 `template` 属性是 V，也就是视图，`title` 属性是 M，也就是模型，这个概念是必须要了解的。

1.2 数据绑定

我们需要一个稍微复杂一点的数据模型来表述 `Todo`，它的结构应该是这样的：

```
{
  value: '事项 1', // 待办事项的文字内容
  done: false    // 标记该事项是否已完成
}
```

由于是多个事项，那么这个数据模型应该是一个数组，为了能先显示这些待办事项，我们需要先设定一些样本数据。在 `Vue` 实例定义中的 `data` 属性中加入以下代码：

```
export default {
  data () {
    return {
      title: 'vue-todos',
      todos: [
```

```

    { value: "阅读一本关于前端开发的书", done: false },
    { value: "补充范例代码", done: true },
    { value: "写心得", done: false }
  ]
}
}
}

```

初学者可能会问 data 有什么作用？我们可以将 Vue 实例定义看作一个类的定义，data 相当于这个类的内部字段属性的定义区域。在 Vue 实例内的其他地方可以直接用 this 引用 data 内定义的任何属性，比如 this.title 就是引用了 data.title。

我们要显示 todos 的数据就需要使用 Vue 模板的一个最常用的 v-for 指令标记，它可以用于枚举一个数组并将对象渲染成一个列表。这个指令使用与 JS 类似的语法对 items 进行枚举，形式为 item in items，items 是数据数组，item 是当前数组元素的别名：

```

<ul>
  <li v-for="todo in todos">
    <label>{{ todo.value }}</label>
  </li>
</ul>

```

它的输出结果如下所示。

```

<ul>
  <li>
    <label>阅读一本关于前端开发的书</label>
  </li>
  <li>
    <label>补充范例代码</label>
  </li>
  <li>
    <label>写心得</label>
  </li>
</ul>

```

如果我们要输出待办事项的序号，可以用 v-for 中隐藏的一个 index 值来进行输出，具体用法如下：

```

<ul>
  <li v-for="(todo,index) in todos"
    :id="index">
    <label>{{ index + 1 }}.{{ todo.value }}</label>
  </li>
</ul>

```

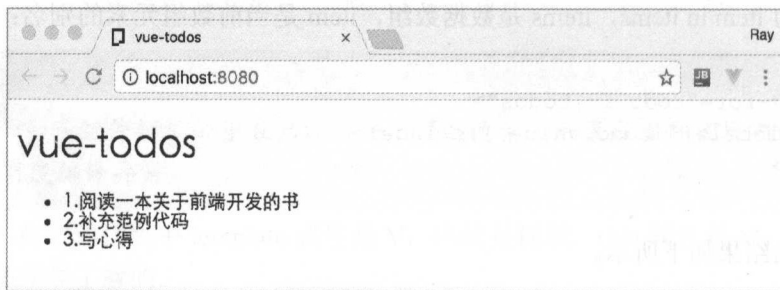
这个用法有点像 Python 的元组引用方式，只要用括号括住引用参数，最后一个值就是循环的索引。索引是由 0 开始计数的，而我们要输出的序号应该从 1 开始，正好我们使用了一个 JavaScript 的表达式插值来输出一个 $\text{index} + 1$ 的从 1 开始计数的序号。

这里除了用插值绑定，还使用了属性绑定语法，就是上面的 `id="index"`，这样的写法是一种缩写，下文中会有解释，意思是将 `index` 的值输出到 DOM 的 `id` 属性上。如果 `index=1`，那么输出结果就是 `id="1"`，如果没有在 `id` 前面加上 `:`，那么 Vue 就会认为我们正在为 `id` 属性赋予一个字符串。

完成这一步，我们打开终端输入：

```
$ npm run dev
```

npm 将自动打开流浏览器并显示以下的结果：



`v-for` 不单单可以循环渲染数组，还可以渲染对象属性，例如：

```
<ul>
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
data () {
  return {
    object {
      first_name : "Ray",
      last_name : "Liang"
    }
  }
}
```

输出

- "Ray"
- "Liang"

小结

对于从来没有接触过 Angular 和 Vue 的初学者，可能对上述的代码感到疑惑，为什么我们的代码内没有任何一个地方操作 DOM 并且将 data 内的变量设置到 DOM 上面呢？

首先，在 Vue 的代码中直接操作 DOM 是不被推荐的，如果你之前是 jQuery 的开发者，这一点一定要牢记；其次，DOM 是被 Vue 直接托管的，所有“绑定”到 DOM 上的变量一旦发生变化，DOM 所对应的属性就会被 Vue 自动重绘而不需要像 jQuery 那样通过编码来显式地操作，这才是绑定的意义所在。

1.3 样式绑定

没有样式的输出结果样子很丑，此时我们就需要用 CSS 来美化我们的 App。我个人并不推荐直接使用 CSS 语法来编写样式表，因为纯 CSS 的代码量很大，而且需要不断地重复，我很讨厌重复而且对 DRY (Don't Repeat Yourself) 有一种偏执。由于 CSS 总是充满各种不得不重复的写法，所以我更愿意使用 less，以下是安装 webpack 支持 less 编译的包的方法：

```
$ npm i less style-loader css-loader less-loader -D
```

安装完成后在 webpack.config.js 的 modules 设置内加入以下的配置：

```
module : {  
  rules: [  
    // ... 省略  
    {  
      test: /\.less$/,  
      loader: "style!css!less"  
    }  
  ]  
}
```

在/assets/中添加一个 todos.less 文件，并在 App.vue 的组件定义内引入 less 样式表：

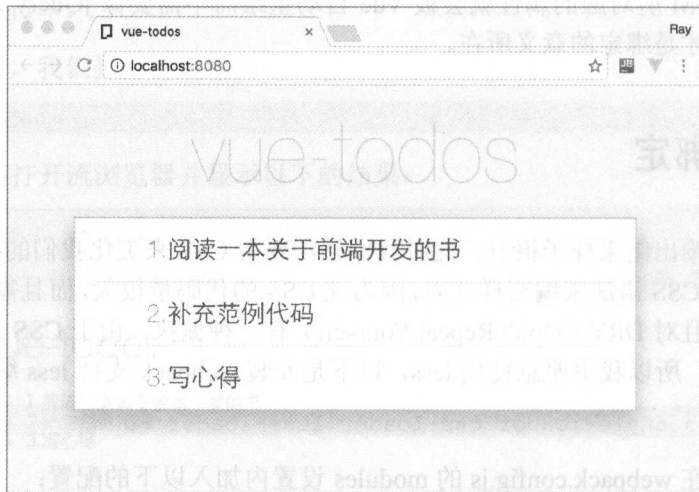
```
import './assets/todos.less'  
export default {  
  // ... 省略  
}
```

使用 import 将样式表直接导入到代码的效果是：webpack 的 less-loader 会生成一些代码，在页面运行的时候将编译后的 less 代码生成到<style>标签内并自动插入到页面的

<head>中。有一点要注意的是，这种做法是全局的，在后面介绍路由部分时会有多个组件页面加载到同一个页上，如果使用 import 导入样式的话，样式就会长期驻留页面直至 Vue 的根（root）实例被销毁。

关于这个 less 样式表的定义属于 HTML 的基础，由于篇幅问题就不在此罗列出来了，读者可以到本书的 github 地址 <http://www.github.com/dotnetage/vue-in-action> 上下载。

运行效果如下：



现在终于舒服多了。这里所有的待办事项都没有显示任何的状态，此时就需要使用 Vue 的样式绑定功能了。

通过 import 将样式文件导入是一种全局性的做法，也就是说，在每一个页面内的<head>中都会有这一个样式表，这样做的缺点是很容易导致样式冲突。如果希望样式表仅应用于当前组件，可以使用<style scoped>，然后用 CSS 的@import 导入样式表：

```
<style scoped>
  @import './assets/todos.less'
</style>
```

前文我们只提到如何将 data 内定义的值以文本插值的方式输出到页面，并没有介绍如何将值“绑定”到属性内。样式的绑定和属性的绑定方式是一样的，我们这里就将 done==true 的待办事项绑定一个 checked 的样式类：

```
<li v-for="(todo,index) in todos"
    :class="{ 'checked': todo.done}" >
  <!-- 省略 ... -->
</li>
```

Vue 的属性绑定语法是通过 `v-bind` 实现的，完整的写法是这样：

```
<li v-for="(todo,index) in todos"
    v-bind:class="{ 'checked': todo.done}">
```

但 `v-bind` 可以采用缩写方式 `“:”` 表示，采用完整写法又将出现各种重复，所以建议还是直接使用缩写方式，这样会更直观。

由此可见，Vue 的属性绑定语法是 `attribute="expression"`，`attribute` 就是元素接收的属性值（既可以是原生的也可以是自定义的），`expression` 则是在 Vue 组件内由 `data` 或 `props` 内定义的对象属性，又或是一个合法的表达式。

要谨记一点：如果在元素属性中不加上 `“:”`，Vue 认为是向这个属性赋上字符串值而不是 Vue 组件上定义的属性引用！

上例中 `:class="{checked: todo.done}"` 的意思是：当 `todo.done` 为 `true` 时，向 `` 元素的 `class` 添加 `checked` 样式类。这是 Vue 样式绑定与普通属性绑定最大的不同点，凡是样式绑定必然是绑定到判断对象上的，不能直接写 CSS 类名，即使要绑定一个固定的 CSS 类也都要这样写，即 `:class="{btn:true}"`，除非不使用样式绑定。

以下是应用样式绑定后的输出效果：



小结

这里推荐一个简单的记忆方法来学习 Vue 的样式绑定，无论绑定的是样式类还是样式属性，`:class` 和 `:style` 表达式内一定是一个 JSON 对象。

- `:class` 的 JSON 对象的值一定是布尔型的, `true` 表示加上样式, `false` 表示移除样式类。
- `:style` 的 JSON 对象则像是一个样式配置项, `key` 声明属性名, `value` 则是样式属性的具体值。

1.4 过滤器

我们在待办事项的右侧增加一个时间字段 `created`, 并用 `<time>` 元素表示, 修改后完整的代码如下所示。

```
<template>
  <div id="app">
    <h1>{{ title }}</h1>
    <ul class="todos">
      <li v-for="(todo,index) in todos"
        :class="{ 'checked': todo.done}">
        <label>{{ index + 1 }}.{{ todo.value }}</label>
        <time>{{ todo.created }}</time>
      </li>
    </ul>
  </div>
</template>
```

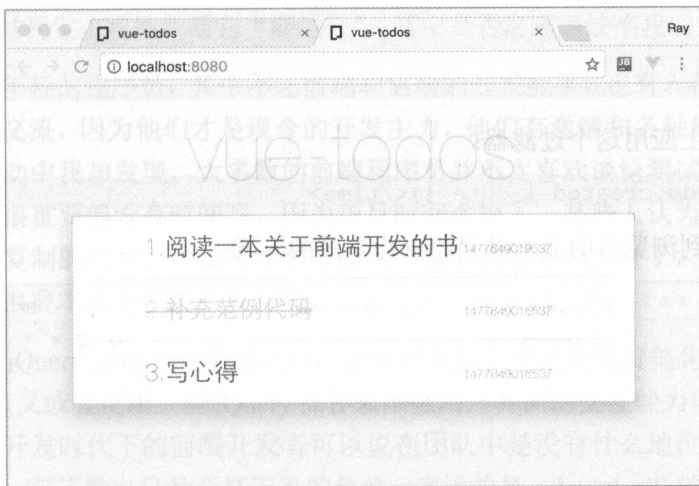
```
<script>
import './assets/todos.less'

export default {
  name: 'app',
  data () {
    return {
      title: 'vue-todos',
      todos: [
        {
          value: "阅读一本关于前端开发的书",
          done: false,
          created: Date.now()
        },
        {
          value: "补充范例代码",
          done: true,
          created: Date.now() + 300000
        },
      ],
    }
  }
}
```



```
{
  value: "写心得",
  done: false,
  created: Date.now() - 30000000
}
]
}
}
}
</script>
```

查看输出结果：



很明显，时间的输出并不是我们想要的结果，这里输出的是一个整数，因为将 `Date` 对象直接输出的话，JavaScript 引擎会将其时间戳作为值输出，所以我们需要对这个时间戳来一个漂亮的格式化。

此时我们可以用一个很出名的时间格式化专用的包——`moment.js`，先安装 `moment.js`：

```
$ npm i moment -S
```

Vue.js 用“过滤器”进行模板格式化，过滤器实质上是一个只带单一输入参数的函数，在 Vue2 中已经将原有的内置过滤器移除了，甚至将一些相关的特色功能也移除了，例如双向过滤器。官方的说法是“计算方法”会比使用“过滤器”更明确，代码更容易读。我认为这有点矫枉过正，过滤器并不是 Vue 和 Angular 这类前端框架所独有的，在很多的服务端视图框架中也是一种很常见的用法。过滤器有用的地方是可以以管道方式进行传递调用。在此，对日期的格式化我还是倾向于使用过滤器的方式，在 Vue 组件中加入自定义过滤器非常简单，只要在 `filters` 属性内加入方法定义就可以在模块上使用了。

首先，我们要引入 `moment`，并设定 `moment` 的区域为中国：

```
import moment from 'moment'
import 'moment/locale/zh-cn'
moment.locale('zh-cn')
```

然后加入一个 `date` 的过滤器：

```
export default {
  // 省略 ...
  filters: {
    date(val) {
      return moment(val).calendar()
    }
  }
}
```

最后在模板上应用这个过滤器：

```
<time>{{ todo.created | date }}</time>
```

我们可以看到浏览器的显示结果将变为下图的方式：



在所有的过滤器中是没有 `this` 引用的，过滤器内的 `this` 是一个 `undefined` 的值，所以不要在过滤器内尝试引用组件实例内的变量或方法，否则会引发空值引用的异常。

第2章 工程化的 Vue.js 开发

我们在上一章中只是用了一个非常简单的例子，尝试对 Vue 的一些基础概念进行大面积的了解，而这个例子也只能跑在开发环境中，它的运行需要有 NPM 和 NPM 上的各种依赖包，所以我们没有办法直接将它放到某个服务器上就能运行，而且它的质量不高。因为在开发过程中我们只是通过视觉主观地判断它“能运行”，其中是否潜藏着缺陷我们并不可知。

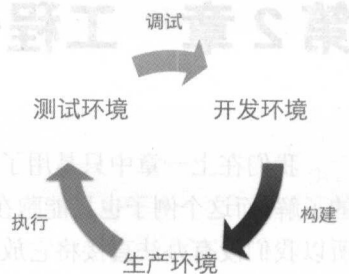
我认识很多年轻的程序员，其中不乏前端和后端的开发能手，也有入行一两年的新手，我很喜欢与他们交流，因为他们才是现今的开发主力，他们有激情和各种标新立异的想法。在各种的交流活动中我却发现，大多数的前端程序员并不太喜欢谈论测试与部署的话题，多数人认为测试很重要但没有时间写，因为项目时间太短了，某些人认为前端程序的部署不过就是个文件复制的过程，只是将本机上的代码复制到服务器上行，让运维人员做就够了，没有必要浪费本就不多的开发时间。

几年前，当 jQuery 还在大行其道之时，前端开发的工作只是负责美化页面样式，修改一些页面的布局，又或者使用一些 jQuery 插件来增强用户界面的交互能力以提升使用体验。可见，处在这种开发时代下的前端开发者可以说在团队中是没有什么地位的，即使有再强的前端开发能力，充其量也只是个打下手的角色。幸运的是，AngularJS 的出现在短短几年间就掀起了一波疯狂的前端技术革命，直到今天这场革命仍然在如火如荼地进行，继而出现了各种形式的响应式编程框架，例如 Polymer、Ember、Knockout、React。呈现着你方唱罢我方上台的格局，当然其中少不了本书的主角 Vue。这波前端的革命浪潮将原本由后端技术所主导的交互页面开发大量地被前端化，前端开发人员的地位也随之水涨船高。能力越大责任越大，这种角色与职责的转化带来的是对前端开发体系、工具甚至是开发方法的一系列改变。在这样的背景下，怎么才能算是一名合格的前端工程师，怎样才能成为一名优秀的前端工程师而不会成为这场风风火火的前端革命中的牺牲者呢？Vue 给了我们一个很好的选择。

选对了前端框架只是第一步，既然前端开发“抢夺”了大量本由后端处理的工作，那么就意味着我们在前端开发过程中要融入一些与后端类似的流程，这样才能真正地达至所谓的“前端开发工程化”。

除了开发，测试与部署可以说是我们在项目开发中的必经阶段，下图很好地诠释了这三者之间的关系。

前端开发与后端开发的不同之处是开放性，换句话说，前端开发具有极强的选择性，工具与开发框架的组合可以说是多得让人眼花缭乱不可胜数。这种开放性带来的多样性选择对于入门者而言无疑像面前横亘了一条深不见底的鸿沟。如果你曾参与过基于 AngularJS 的开发项目，那你必然会体验到搭建一个多人协作的开发、测试和部署环境如身陷泥潭般难以前行；当你进入到 React 项目也会有类似的感觉，不同的只是从工具的复杂性陷阱跳入了第三方依赖包所构建的陷阱。



Vue 作为 AngularJS 和 React 的继承者和改良者，它不单单从编码的特色与开发框架本身的使用上进行大面积的改良与优化。而更实用之处也是它的优秀之处，在于它提供了一整套简化开发、测试与部署的方案。作为它的前端开发者，不再需要花大量的时间去学习理解框架的使用概念，以及耗费大量的精力去建立复杂的自动化环境。

在此我特意以一章的篇幅讲述 Vue 如何为我们建立开发、测试与部署的自动化环境，我们又将如何在它们的基础上有针对性地进行定制与改良。

2.1 脚手架 vue-cli

当我们使用 Vue 构建一个原型的时候，需要做的通常就是通过 `<script>` 把 `Vue.js` 引入进来，然后就可以在页面上直接进行编码了。这种情况仅仅作为一个实验性的尝试完全是可以的，但是真实的开发却不能这样做。

真正在前端开发时，不可避免地要用到一大堆的工具，例如模块化的包管理工具，代码运行前的预处理器，程序热加载模块，代码校验，还有各种的测试环境与框架支持工具等，这些工具对于一个需要长期维护或不断地迭代演进的应用来说都是必需的。从项目初始化开始，安装配置这些不同开发场景下的支撑工具是家常便饭，但却又是非常让人感到痛苦的。这些开发环境的支持工具很多，而且配置方式各不相同，如果没有一定的使用经验，在项目初始化时就配置一个具有良好扩展性的工具环境，这种安装配置的工作就还会不断地重复出现。

我很喜欢 Vue 的一个重要原因就是因为它有 `vue-cli`，这个工具可以让一个简单的命令行工具来帮助我快速地构建一个足以支撑实际项目开发的 Vue 环境，并不像 Angular 和 React 那样要在 Yoman 上找适合自己的第三方脚手架。`vue-cli` 的存在将项目环境的初始化工作与复杂度降到了最低。

安装 vue-cli

`vue-cli` 是一个 npm 的安装包，我们希望它能在本机的任意目录下创建项目，那么就得将它安装到 node.js 的全局运行目录下：

```
$ npm i vue-cli -g
```

安装成功后，我们就可以使用 `vue-cli` 来初始化 Vue 项目了。

使用 vue-cli 初始化项目

`vue-cli` 是一个很简单的指令，先打开它的帮助文件看看它的具体用法：

用法：`vue <命令> [选项]`

命令：

<code>init</code>	从指定模板中生成一个新的项目
<code>list</code>	列出所有的可用的官方模板
<code>help [cmd]</code>	显示所有 <code>[cmd]</code> （命令）的帮助

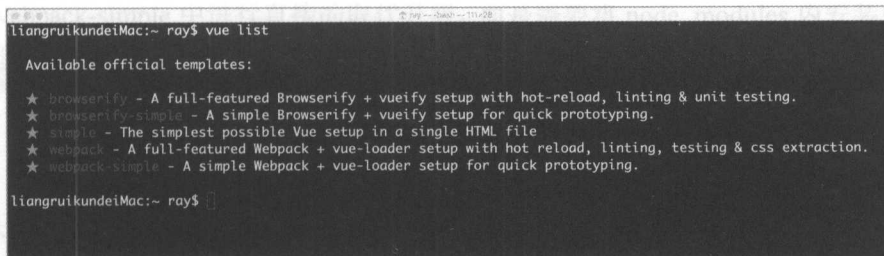
选项：

<code>-h, --help</code>	输出用法信息
<code>-V, --version</code>	输出版本号

先用 `list` 指令来看看有哪些官方模板可用：

```
$ vue list
```

输出结果如下图所示。



```
liangruikundeMac:~ ray$ vue list

Available official templates:

★ browserify - A full-featured Browserify + vueify setup with hot-reload, linting & unit testing.
★ browserify-simple - A simple Browserify + vueify setup for quick prototyping.
★ simple - The simplest possible Vue setup in a single HTML file
★ webpack - A full-featured Webpack + vue-loader setup with hot reload, linting, testing & css extraction.
★ webpack-simple - A simple Webpack + vue-loader setup for quick prototyping.

liangruikundeMac:~ ray$
```

这些官方模板存在的意义在于提供强大的项目构建能力，用户可以尽可能快地进行开发。然而能否真正地发挥作用还在于用户如何组织代码和使用的其他库。

将 list 指令的输出结果翻译一下，就可以清楚地了解这些官方模板应用于哪些使用场景：

- browserify——拥有高级功能的 Browserify + vueify 用于正式开发；
- browserify-simple——拥有基础功能的 Browserify + vueify 用于快速原型开发；
- simple——适用于单页应用开发的最小化配置；
- webpack——拥有高级功能的 webpack + vue-loader 用于正式开发；
- webpack-simple——拥有基础功能的 webpack + vue-loader 用于快速原型开发。

browserify 的模板做得比较简陋，就算是用于正式开发还是会有些不足，配置的是 Karma+Jasmine 的单元测试框架，而 browserify 属于比较老旧的构建工具，估计官方提供这两个模板页是出于对经常使用 browserify 的开发人员提供一个熟悉环境的考虑。到了正式的项目开发时，我们还是会走上 webpack 的道路。

所以我建议初学者可以跳过 browserify 的两个模板，直接使用 webpack 的两个模板。首先 webpack-simple 正如其名，配置了最简单的可直接支持 ES6 的 Vue.js 编译环境，可以应对那些要求时间短，结构相对简单的小型应用。如果对所有环境工具都非常熟悉，开发者也可以由这个模板入手，为项目底板定制更适应自身开发要求的环境。

其次，webpack 模板是一个非常赞的脚手架，将其分析透彻之后，就会知道 Vue 的官方开发团队在其中花了很大的功夫，将上文所叙述的开发、测试与生产环境做了非常完善的配置，从最大程度上简化了由于工具而引入项目的复杂度，也降低了开发人员对工具的学习成本，这个模板也将是本书中讲述的重点。

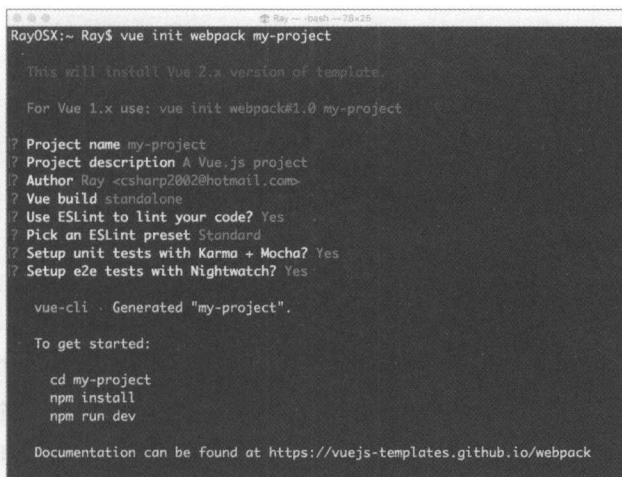
创建项目

接下来先看看这个 vue-cli 如何为我们创建项目。创建项目使用的是 init 命令，它会为我们自动创建一个新的文件夹，并将所需的文件、目录、配置和依赖都准备好，具体做法如下：

```
$ vue init webpack my-project
```

init 命令执行后会出一系列的交互式问题让我们选择，运行结果如下所示。

完成以后直接按提示进入项目，安装 npm 的依赖包后就可以开始开发。



```
RayOSX:~ Ray$ vue init webpack my-project

This will install Vue 2.x version of template.

For Vue 1.x use: vue init webpack@1.0 my-project

? Project name my-project
? Project description A Vue.js project
? Author Ray <csharp2002@hotmail.com>
? Vue build standalone
? Use ESLint to lint your code? Yes
? Pick an ESLint preset Standard
? Setup unit tests with Karma + Mocha? Yes
? Setup e2e tests with Nightwatch? Yes

vue-cli · Generated "my-project".

To get started:

  cd my-project
  npm install
  npm run dev

Documentation can be found at https://vuejs-templates.github.io/webpack
```

2.2 深入 vue-cli 的工程模板

vue-cli 提供的脚手架只是一个最基础的，也可以说是 Vue 团队认为的工程结构的一种最佳实践。对于初学者或者以前曾从事 AngularJS/React 开发的用户来说，可能对开发环境有自己习惯性用法和熟悉的工具，但我建议用 Vue 来开发的话还是先按照官方推荐的来做，待我们掌握了 Vue 官方推荐的环境配置后再按照实际情况进行相应的调整，这样会少走一些弯路，节省不少时间。

我们下面要讨论的工程结构都是围绕 webpack-simple 与 webpack 展开的，browserify 也只是在这两个模板的基础上移植的一个版本，所以就不过多地赘述。

webpack 和 webpack-simple 这两个模板从文件结构上看几乎是一致的，只是一个简化版，另一个是完全版。其实不然，webpack-simple 是基于 Webpack@2.1.0-beta.25 进行配置的版本，而 webpack 模板则是基于 Webpack ^1.3.2 配置的。这两个版本暂时是互不兼容的，而且使用的依赖包的版本也不一样，所以不要将 webpack 模板创建的项目文件结构复制到 webpack-simple 中进行直接的取代升级，而是需要将 node_modules 内安装的所有的依赖包删除，然后重新安装才有可能迁移成功，这一点是需要注意的。

2.2.1 webpack-simple 模板

以下为 webpack-simple 模板构建的项目的工程目录结构：

```
.
├── README.md
├── index.html
├── package.json
├── src
│   ├── App.vue
│   ├── assets
│   │   └── logo.png
│   └── main.js
└── webpack.config.js
```

webpack-simple 只配置了 Babel 和 Vue 的编译器，其他的一无所有。这个模板值得一提的就是 src 目录，所有的 Vue 代码源程序都放置在这个目录中，五个模板构建出来的这个 src 目录都是一样的，只是在 webpack 模板中多了 components 目录用于存放公用组件。这个目录的结构与文件的组织应在开发前就进行约定，对于多人协作式项目，目录的使用与文件的命名都显得尤为重要。

具体约定如下：

(1) 公共组件、指令、过滤器（多于三个文件以上的引用）将分别存放于 src 目录下的

- components;
- directives;
- filters。

(2) 以使用场景命名 Vue 的页面文件。

(3) 当页面文件具有私有组件、指令和过滤器时，则建立一个与页面同名的目录，页面文件更名为 index.vue，将页面与相关的依赖文件放在一起。

(4) 目录由全小写的名词、动名词或分词命名，由两个以上的词组成，以“-”进行分隔。

(5) Vue 文件统一以大驼峰命名法命名，仅入口文件 index.vue 采用小写。

(6) 测试文件一律以测试目标文件名.spec.js 命名。

(7) 资源文件一律以小写字符命名，由两个以上的词组成，以“-”进行分隔。

例如：

```
src
├── README.md
├── assets           // 全局资源目录
└── images          // 图片
```



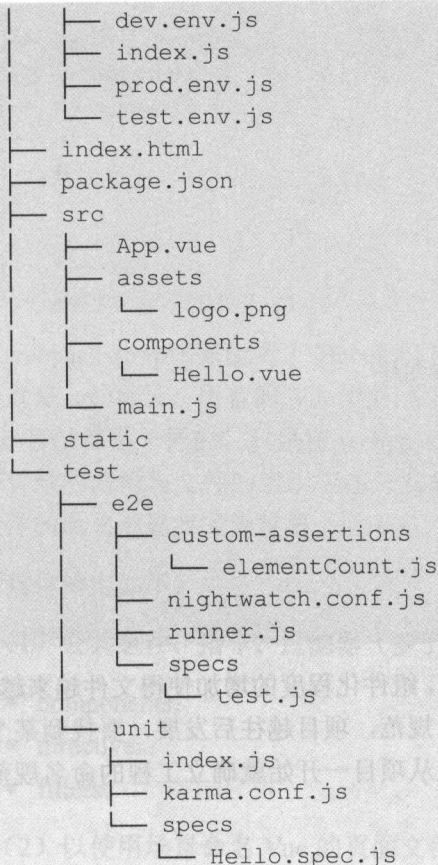
```
| | less          // less 样式表
| | css          // CSS 样式表
| | fonts        // 自定义字体文件
| | components   // 公共组件目录
| |   ImageInput.vue
| |   Slider.vue
| |   ...
| | directives.js // 公共指令
| | filters.js    // 公共过滤器
| | login         // 场景：登录
| |   index.vue   // 入口文件
| |   LoginForm.vue // 登录场景私有表单组件
| |   SocialLogin.vue
| | cart
| |   index.vue
| |   ItemList.vue
| |   CheckoutForm.vue
| | Discover.vue  // 场景入口文件
| | App.vue      // 默认程序入口
| | main.js
```

前端开发的文件非常零碎而且会随着项目增多，组件化程度的增加使得文件越来越多，如果从一开始就没有约定目录与文件的使用与命名规范，项目越往后发展，要找到某个文件就越困难，各种古怪的名字也会随之显现，所以从项目一开始就确立工程的命名规范与使用约定是很有必要的。

2.2.2 webpack 模板

webpack 模板的工程目录结构如下：

```
| | README.md
| | build
| |   build.js
| |   check-versions.js
| |   dev-client.js
| |   dev-server.js
| |   utils.js
| |   webpack.base.conf.js
| |   webpack.dev.conf.js
| |   webpack.prod.conf.js
| | config
```



是不是觉得这个工程结构非常复杂？第一次使用的时候我也顿生此感，但这个 webpack 模板的结构是非常合理的，而且配置的工具也相当丰富，当投入真正的项目开发时会觉得模板的实用性很强。

所以我们很有必要花些时间将这个模板的结构以及它所提供的工具配置了解清楚，掌握 Vue 官方团队对项目开发的环境配置与使用思路，以便于我们能结合自己的实际情况进行适当的配置与调整。

在上文中我们已经提过 `src` 目录的用法与约定，此处就不再赘述。在项目的根目录下多了4个目录，它们的作用分别如下：

- **build**——存放用于编译用的 **webpack** 配置与相关的辅助工具代码；
- **config**——存放三大环境配置文件，用于设定环境变量和必要的路径信息；
- **test**——存放 E2E 测试与单元测试文件以及相关的配置文件；

- `static`——存放项目所需要的其他静态资源文件;
- `dist`——存放运行 `npm run build` 指令后的生产环境输出文件, 可直接部署到服务器对应的静态资源文件夹内, 该文件夹只有在运行 `build` 之后才会生成。

可见, 这些目录的存在是依赖于模板内配置的开发工具的, `webpack` 模板配置以下的工具。

2.2.3 构建工具

由于开发、测试与生产三大运行环境都需要进行构建, 而且针对不同的环境要求, 它的配置会有一些区别, 本书后面的章节中我们会对具体的配置进行一些定制与修改, 我们应该清楚地了解 `webpack` 模板是如何进行构建的。

1. 编译开发环境

在开发环境下通过以下指令加载运行 `Vue` 项目:

```
$ npm run dev
```

这个指令的配置是在 `package.json` 的 `script` 属性中设置的, 实质上它是由 `npm` 来引导执行入口程序 `dev-server.js` 完成以下的加载过程:



加载环境变量

该环节从 `config` 目录加载 `index.js` 和 `dev.env.js` 两个模块, 准备开发调试环境所必需的一些目录和全局变量。

合并 webpack 配置

在 `build` 目录下一共有三个 `webpack` 的配置文件:

- `webpack.base.conf.js`——公用的基本 `webpack` 配置;
- `webpack.dev.conf.js`——开发环境专用的 `webpack` 配置项;
- `webpack.prod.conf.js`——生产环境专用的 `webpack` 配置项。

这里使用了一个叫 `webpack-merge` 的包来进行两个 `webpack` 配置之间的合并, 这个环节就是通过这个包将 `webpack.base.conf.js` 和 `webpack.dev.conf.js` 合并成最终的 `webpack` 配置。

请记住这几个配置文件, 在下面的章节中我们会对这些配置的内容进行调整。

配置热加载

热加载是一个非常棒的功能，这个功能启用后的效果就是：当开发环境被启动并进入调试模式后，一旦我们修改了任意地方的源代码，浏览器中对应的内容就会被自动刷新，而无须手工对浏览器进行刷新的操作，这个配置将是我们做页面布局或者功能调整时的一大臂助。

上一个环境中合并的 webpack 配置也是通过这个环节被动态加载的，当代码文件发生变化，热加载就会启动 webpack 进行重新编译，然后将最新的编译文件重新加载到浏览器中。

配置代理服务器

这个环境是为我们的代码增加一个模拟的服务端做准备，有了它的存在，我们就可以在没有后端程序支持的情况下，直接模拟远程服务器执行的一些请求的效果。例如，向服务器发出一个 HTTP GET /api/books/ 的请求，那么我们就可以利用代理服务器将这一请求截获下来，然后返回一组这个 API 应该执行成功的返回结果，这样我们的前端程序运行起来的效果就与接入了服务端后的效果是一致的了。我们将这一技术称为服务模拟，在后面的章节中会具体介绍这一技术。

配置静态资源

将图片、字体、样式表和编译后的 JS 脚本等，生成对应的一些印记（Footprint）并存放由开发服务器托管的一个 static 虚目录中，使得我们在浏览器中可以正常访问到这些资源。每个生成的文件 Footprint 是一些哈希代码，当文件内容发生变化时这些哈希代码就会发生改变，使用 Footprint 是将静态文件发布到 CDN 或者进行离线缓冲时通知浏览器文件是否发生改变的重要依据。

加载开发服务器

启动一个 Express 的 Web 服务器，将上述各个环境中配置好的模块进行加载，并使程序能通过浏览器进行访问。

以上就是 npm run dev 的完整执行思路。

2. 编译生产环境

当项目准备发布时，在命令行键入：

```
$ npm run build
```

执行效果如下：

```

liangruikundeMac:vue-grid ray$ npm run build
> vue-grid@ build /Users/ray/projects/vue-grid
> node build/build.js

Tip:
Built files are meant to be served over an HTTP server.
Opening index.html over file:// won't work.

building for production...cp: no such file or directory: static/*
Hash: 7cb8d6feec3fa2e09ea1
Version: webpack 1.13.3
Time: 19774ms

  Asset      Size  Chunks  Chunk Names
static/css/app.9fdb8757462ed4ee47d09312a74aaa9b.css  221 kB  2, 0    [emitted]  app
static/fonts/fontawesome-webfont.1dc35d2.ttf        153 kB          [emitted]
static/fonts/fontawesome-webfont.e6cf7c6.woff2      71.9 kB          [emitted]
static/js/manifest.ce76013dabdbffad6282.js          832 bytes  0    [emitted]  manifest
static/js/vendor.c962c57d67115197c61a.js           500 kB  1, 0    [emitted]  vendor
static/js/app.6d2e93da915d1c32adca.js               28.8 kB  2, 0    [emitted]  app
static/fonts/fontawesome-webfont.c8ddf1e.woff       90.4 kB          [emitted]
static/js/manifest.ce76013dabdbffad6282.js.map       8.86 kB  0    [emitted]  manifest
static/js/vendor.c962c57d67115197c61a.js.map         4.15 MB  1, 0    [emitted]  vendor
static/js/app.6d2e93da915d1c32adca.js.map            127 kB  2, 0    [emitted]  app
static/css/app.9fdb8757462ed4ee47d09312a74aaa9b.css.map 486 kB  2, 0    [emitted]  app
index.html      489 bytes          [emitted]

liangruikundeMac:vue-grid ray$

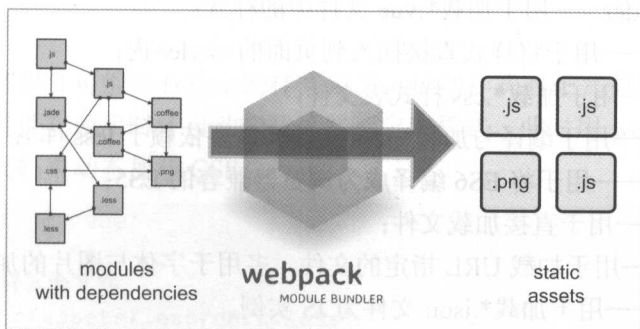
```

生产环境的构建过程比较简单，首先是对必要的资源文件进行打包加上 FootPrint，然后是对脚本进行编译、压缩和包大小的分割。

2.3 Vue 工程的 webpack 配置与基本用法

我们在真实的 Vue 项目开发过程中，会因为很多不同的实际运用需求不断地对 webpack 配置进行修改，在此之前，我们需要对 webpack 有一个基本的认识，了解它到底能为我们做些什么。

webpack 是一个模块打包的工具，它的作用是把互相依赖的模块处理成静态资源，如下图所示。



现有的模块打包工具不适合大型项目（大型的 SPA）的开发。当然最重要的还是因为缺少代码分割功能，以及静态资源需要通过模块化来无缝衔接。webpack 的作者曾经试图对原有的打包工具进行扩展，但是没能成功。webpack 的目标：

- 把依赖树按需分割；
- 把初始加载时间控制在较低的水平；
- 每个静态资源都应该成为一个模块；
- 能把第三方库集成到项目里成为一个模块；
- 能定制模块打包器的每个部分；
- 能适用于大型项目。

2.3.1 webpack 的特点

代码分割

在 webpack 的依赖树里有两种类型的依赖：同步依赖和异步依赖。异步依赖会成为一个代码分割点，并且组成一个新的代码块。在代码块组成的树被优化之后，每个代码块都会保存在一个单独的文件里。

加载器

webpack 原生是只能处理 JavaScript 的，而加载器的作用是把其他的代码转换成 JavaScript 代码，这样一来所有种类的代码都能组成一个模块，也就是说，我们可以在代码内通过 import 将 webpack 打包的资源以模块的方式引入到程序中。

以下是 Vue 项目中常用到的加载器（它们都是以 NPM 库形式提供的）：

- vue-loader——用于加载与编译*.vue 文件；
- vue-style-loader——用于加载*.vue 文件中的样式；
- style-loader——用于将样式直接插入到页面的<style>内；
- css-loader——用于加载*.css 样式表文件；
- less-loader——用于编译与加载*.less 文件（需要依赖于 less 库）；
- babel-loader——用于将 ES6 编译成为浏览器兼容的 ES5；
- file-loader——用于直接加载文件；
- url-loader——用于加载 URL 指定的文件，多用于字体与图片的加载；
- json-loader——用于加载*.json 文件为 JS 实例。

智能解析

webpack 的智能解析器能处理几乎所有的第三方库，它甚至允许依赖里出现这样的表达式：

```
require("../components/"+ name + ".vue")
```

这一点恰恰是 browserify 不能做到的。

它能处理大多数的模块系统，比如说 CommonJS 和 AMD。

插件系统

webpack 有丰富的插件系统，大多数内部的功能都是基于这个插件系统的。这也使得我们可以定制 webpack，把它打造成能满足我们需求的工具，并且把自己做的插件开源出去。

2.3.2 基本用法

webpack 的打包依赖于它的一个重要配置文件 webpack.config.js，在这个配置文件中就可以指定所有在源代码编译过程中的工作了！对，就一个配置就可以与冗长的 Gruntfile 或者 Gulpfile 说再见了。

下面就可直接完成打包的工作了，通过这样精简化的配置是不是感觉已经了解了 webpack？当然一个完整的工程项目中的 webpack 的配置远远没有这么简单，随着工程的构建要求的增加，webpack.config.js 内的配置项目也会随之增加，webpack 还有许许多多的选项提供给我们进行灵活配置，但这不是本书最重要的内容，它只是一个构建工具，我们只需要了解在 Vue 项目中它基本能为我们做到的工作、最小化的配置是如何的就足够了，在以后需要对它进行扩展与优化时，带着问题去查官方文档也是非常容易的事。

样式表引用

某些页面或者组件可能具有特定的样式定义，这些样式对于其他页面来说是冗余的，我们只希望这些组件在应用时才自动加载这些特定的样式，此时用 webpack 我们就能在源代码中加入以下代码来动态加载 CSS：

```
import Vue from 'vue'  
// ... 省略  
// 引用指定的样式源文件  
import './app/assets/less/dark.less'
```

```
export default {
  // ... 省略
}
```

此时我们只需要在 webpack 的配置中加入 less-loader，那么 webpack 在打包的时候就会自动将 less 转换为 CSS，并将 CSS 的动态代码生成到 JS 文件中。当 Vue 组件被加载到页面并实例化后，将在 DOM 内插入这个特定的行内样式<style>以实现动态样式的应用。

对于*.css 文件同样也是适用的，例如导入某个第三方库中必需的样式表：

```
import 'uikit/dist/css/components/tabs.css'
```

字体的引用

假设在 dark.less 内加入对自定义字体文件的样式定义：

```
@font-face {
  font-family: 'Darkenstone';
  src: url('./Darkenstone.eot');
  src: url('./Darkenstone.eot?#iefix') format('embedded-opentype'),
    url('./Darkenstone.woff2') format('woff2'),
    url('./Darkenstone.woff') format('woff'),
    url('./Darkenstone.ttf') format('truetype'),
    url('./Darkenstone.svg#Darkenstone') format('svg');
  font-weight: normal;
  font-style: normal;
}

.header
{
  display: flex;
  flex-flow: row nowrap;

  & > h1 {
    font: 16pt 'Darkenstone';
  }}
}
```

这里.header>h1 指定了一个 Darkenstone 的自定义字体，这个字体浏览器一定是不能识别的，以前我们在样式表中先定义这个字体样式并指定加载位置（如上文@font-face 的定义），然后在页面中引用这个样式表，这是多么麻烦的一件事，不是吗？

如果用了 webpack 后，我们只是在配置文件内加入了一个 url-loader：

```
{
  test: /\. (woff2?|eot|ttf|otf) (\?.*)?$/ ,
```

```
loader: 'url'
}
```

我们并不需要在源代码中做任何改变，因为之前已经引用过样式表 `dark.less`，而字体是在样式表中的，`webpack` 将在打包的时候为我们识别并在代码中引入字体的动态加载。这样一来极大地解决了我们对资源引用的依赖问题！

`vue-cli` 的 `webpack` 模板已经为我们配置好了绝大多数常用的 loader，在实际运用中我们只需要了解它们是怎么来的，应该怎么用，需要的时候如何修改就够了。

2.3.3 用别名取代路径引用

在项目开发过程中有可能有许多包是没有放在 `npm` 上的，有一些较老的可能还依然只存在于 `bower` 上，某些甚至在 `bower` 与 `npm` 上都找不到，而不得不通过下载的方式在项目内引用，这样一来我们的代码可能通过 `require` 就得在代码内引用一段很长的文件路径，如下所示。

```
import Selector from '../..../bower_components/bootstrap-select/dist/js/select'
```

这种包的引用方式明显违反了 `CommonJS` 的编程规范，对于这些长路径，甚至还具有“`../..`”这些相对路径搜索的定义，我们可以通过 `webpack` 的 `resolve` 配置项来解决。就以 `select` 这个组件为例，在 `webpack.base.config.js` 中加入以下的这个别名的定义：

```
module.exports = {
  entry: { ... },
  output: { ... },
  module: { ... },
  resolve: {
    extensions: ['', '.js'],
    alias: {
      'bs-select':
        'bower_components/bootstrap-select/dist/js/select.js'
    }
  }
}
```

有了这个定义以后，我们就可以将上面那个长引用改为下面的写法：

```
import Selector from 'bs-select';
```

绝对不要让路径引用进入到我们的代码，因为这是代码的“癌症”，一旦开始植入并生长起来，以前的代码将难以维护！

2.3.4 配置多入口程序

多数情况下我们的程序入口不单单只有一个，举一个最简单的例子，前台提供给最终用户使用（<http://www.domain.com/index>），后台提供给登录用户使用（<http://www.domain.com/admin/>），那么自然需要多个与 main.js 类似的程序入口了。

首先在 build/webpack.base.conf.js 配置文件中的 entry 配置属性上加上新的入口文件：

```
module.exports = {
  entry: {
    app: './src/main.js',
    admin: './src/admin-main.js'
  },
  // ... 省略
}
```

这是用于告诉 webpack 哪几个是入口文件，这些文件需要被生成到启动页的<script>内。

vue-cli 的 webpack 模板使用 HtmlWebpackPlugin 插件，生成 HTML 入口页面并自动将生成后的 JS 文件和 CSS 文件的引用地址写入到页内的<script>中。

这里就需要在 build/webpack.dev.config.js 文件内的 plugins 配置项内多配置一个 HtmlWebpackPlugin 插件，用于生成 admin.html 入口页。

```
plugins:[
  // ... 省略

  // 这是原有的配置项，用于匹配注入 app.js 的输出脚本
  new HtmlWebpackPlugin({
    filename: process.env.NODE_ENV === 'testing'
      ? 'index.html'
      : config.build.index,
    template: 'index.html',
    chunks: ['app'], // 与原配置的不同的是要用 chunks 指定对应的 entry
    inject: true,
    minify: {
      removeComments: true,
      collapseWhitespace: true,
      removeAttributeQuotes: true
    },
    chunksSortMode: 'dependency'
  }),
]
```

```
// 这是新增项，用于匹配注入 admin.js 的输出脚本
new HtmlWebpackPlugin({
  filename: process.env.NODE_ENV === 'testing'
    ? 'admin.html'
    : config.build.admin,
  template: 'index.html',
  chunks: ['admin'],
  inject: true,
  minify: {
    removeComments: true,
    collapseWhitespace: true,
    removeAttributeQuotes: true
  },
  chunksSortMode: 'dependency'
}),
]
```

需要强调一点的是，这里的 `HtmlWebpackPlugin` 配置必须用 `chunks` 指定在上文 `entry` 内对应的入口文件的别名。

关于 `HtmlWebpackPlugin` 更多配置内容可以参考：<https://github.com/kangax/html-minifier#options-quick-reference>。

还有就是得将同样的配置加入到生产环境专用的 `webpack` 配置文件 `webpack.prod.conf.js` 中，否则当我们运行 `npm run build` 时是不会输出 `admin.js` 和 `admin.html` 这两个入口文件的（由于配置内容相同这里就不再重复了）。

最后，如果使用了 `vue-router` 就得对 `connect-history-api-fallback` 插件的配置进行修改，否则原有的默认配置只会将所有的请求转发给 `index.html`，这样就会导致 `History API` 没有办法正确地将请求指向 `admin.html`，导致热加载失败，具体做法如下所述。

打开 `dev-server.js` 文件，将 `app.use(require('connect-history-api-fallback')())` 配置改为以下的方式：

```
// handle fallback for HTML5 history API
var history = require('connect-history-api-fallback')
// app.use(require('connect-history-api-fallback')())

app.use(history({
  rewrites: [
    { from: /^\/admin\/.*$/, to: '/admin.html' }
  ]
})
```



```
});
```

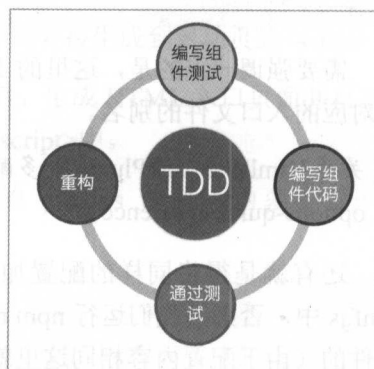
新入口需要有明确区分的路由规则，否则还是会产生热加载失败的情况，这样就非常不利于开发了。

2.4 基于 Karma+Phantom+Mocha+Sinon+Chai 的单元测试环境

一般来说，Vue 项目的单元测试用得最多的就是组件功能测试了。具体如何来写这些组件测试将在第 5 章讲解。在此，我们先从开发流程的角度来看待测试的问题以及深入了解 vue-cli webpack 模板为我们建立的单元测试环境的功能与运作机理。

首先，在有具体的组件测试目标时的 Vue-TDD 的开发流程如下图所示。

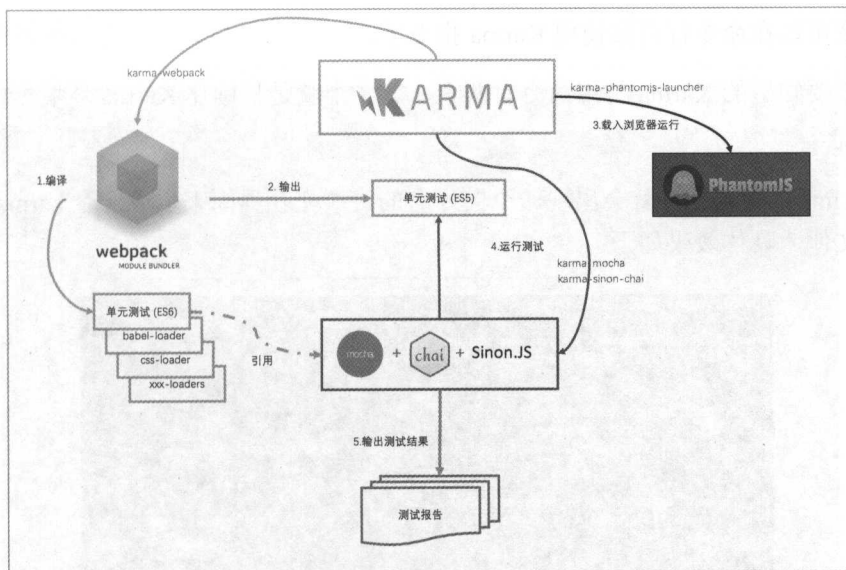
- 编写组件测试——在单元测试中将设计组件的名称、属性接口、事件接口，用断言工具确定衡量这个组件正确的标准。
- 编写组件代码——以单元测试作为引导程序，编写组件真实的实现代码，让测试通过。
- 运行测试，并看到测试通过。
- 重构。



然后如此循环直至所有的组件单元测试都通过为止。与运行 `npm run dev` 指令将代码加载到浏览器中运行不同的是，这个过程并不需要我们打开浏览器用眼睛判断组件的输出是否符合要求，用调试模式来观察变量是否正确，因为这一切都应该是自动执行的！vue-cli 的 webpack 模板就为我们配置了这样一个全自动化的测试环境，作为高质量 Vue 组件开发的最大助力。

前端开发的单元测试环境会比后端开发的单元测试环境复杂，这是由于前端开发的工具碎片化比较严重所致的，所以要配置一个良好的单元测试环境需要有长期的实战经验以及熟悉各种各样的工具，vue-cli 的 webpack 模板给我们配置的单元测试环境其实也相当复杂，下图描绘了这些工具是如何进行协作的。

接下来我们就一个一个地了解这些工具的作用，以便我们在做单元测试的时候知道可以对哪些环节进行调整和优化。



Karma

Karma 是一个著名的测试加载器 (<https://karma-runner.github.io/>)，它能完成许多测试环境加载任务。

Karma 作为自动化测试程序的入口，它可以执行以下这些任务：

- 为测试程序注入指定依赖包；
- 可同时在一个或多个浏览器宿主中执行测试，满足兼容性测试需要；
- 执行代码覆盖性测试；
- 输出测试报告；
- 执行自动化测试。

Karma 就是这样一个开发环境，开发者指定需要测试的脚本/测试文件，需要运行的浏览器等信息，Karma 会在后台自动监控文件的修改，并启动一个浏览器与 Karma 的服务器连接，这样当源代码或者测试发生修改后，Karma 会自动运行测试。

开发者可以指定不同的浏览器，甚至可以跨设备。由于 Karma 只是一个运行器，所以要配置一些测试框架如 Mocha、Jasmine 等作为单元测试的代码支撑，甚至还可以自定义适配器来支持自己的测试框架。

Karma 拥有独立的 CLI，可以通过以下方式安装到全局环境中：

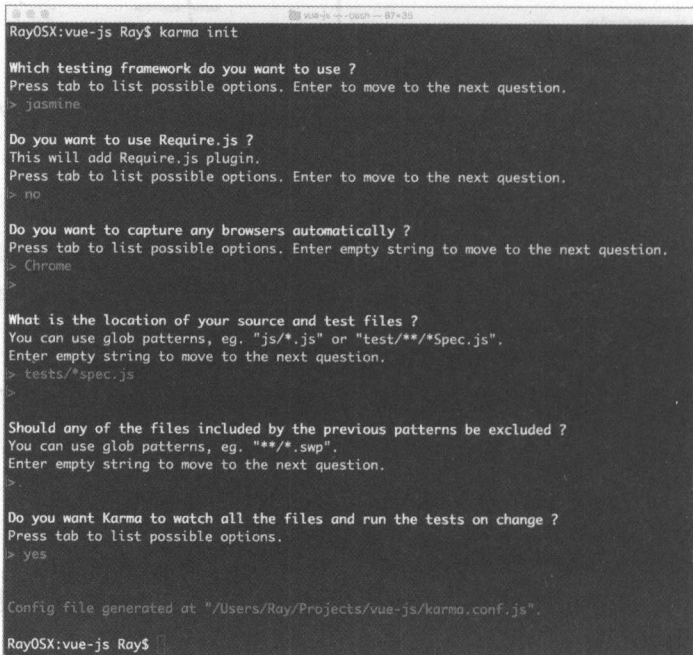
```
$ npm karma i -g
```

然后就可以在命令行直接使用 Karma 指令了。

我们需要知道的 Karma 常用命令有两个，第一个就是初始化 Karma 环境：

```
$ karma init
```

karma init 指令运行后就会出现一个向导型的终端交互界面来生成一个 karma.conf.js 的全局配置文件，具体效果如下：



```
RayOSX:vue-js Ray$ karma init
Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine
Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no
Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*.spec.js".
Enter empty string to move to the next question.
> tests/*spec.js
Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>
Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> yes
Config file generated at "/Users/Ray/Projects/vue-js/karma.conf.js".
RayOSX:vue-js Ray$
```

如果正在使用 vue-cli webpack 模板就不需要手工来做这一步，因为模板在创建工程时就生成了这个配置文件，在 `~/test/unit/karma.conf.js` 中就可以找到它。

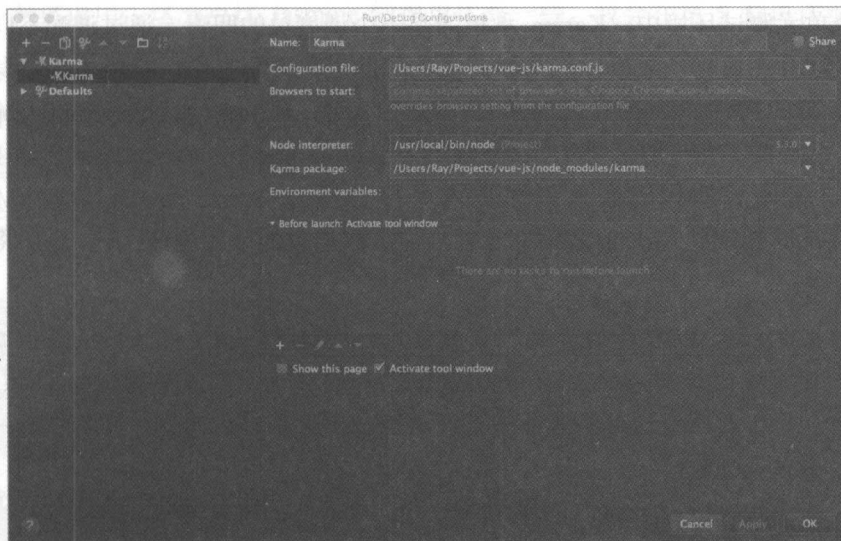
第二个命令就是 `karma start`，这个命令就是启动 Karma，让它按照 `karma.conf.js` 的配置项执行自动化测试。vue-cli webpack 模板将这个指令在 `package.json` 内进行了包装定义，所以在工程目录下只要运行：

```
$ npm run unit
```

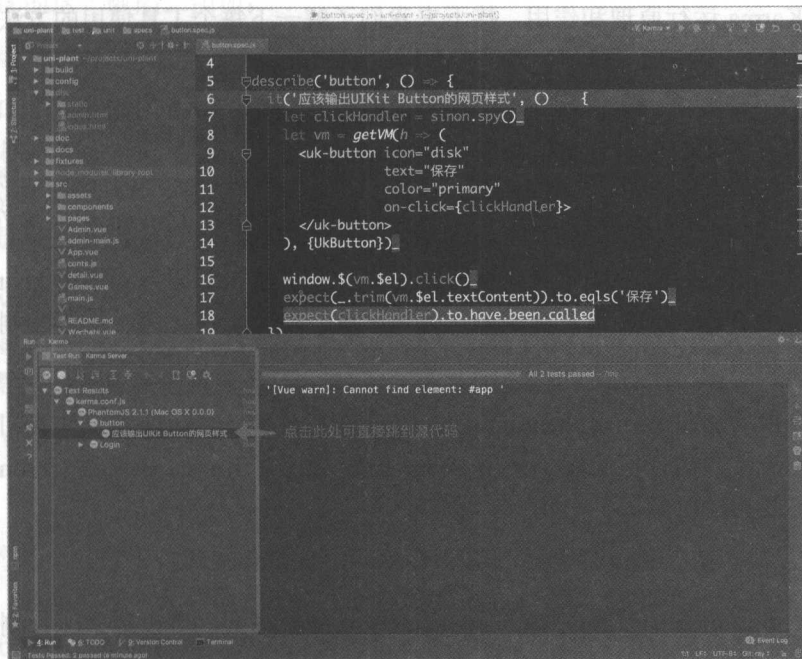
就可以直接启动 Karma。

另外，Karma 还能很好地与 WebStorm 集成在一起，只要在 WebStorm 的“Run/Edit Configurations”菜单中打开运行器对话框，然后增加一个 Karam 的运行项，在“Configuratio File”选择框内找到当前工程的 `karma.conf.js`，就可以在 WebStrom 内直接运行 Karma 了，

配置如下图所示。



在 WebStorm 内直接运行 Karma，WebStorm 会将单元测试项集成在 IDE 内，而不单单只是在终端输出 Karma 的测试结果报告：



Karma 的插件系统

在整个单元测试环境中，Karma 承担的是一个调度员的职责，通过调配不同的工具让其有条不紊地互相协作。之所以能与如此多的外部工具协同工作，是由于它自身强大的插件系统和丰富的插件库。在我们的单元测试环境中，Karam 通过 karma-webpack 启动 webpack 编译测试文件与源代码文件。然后通过 karma-phantomjs-lanucher 启动 PhantomJS 浏览器，将编译后的代码嵌入到网页内。接着通过 karma-mocha 启动 Mocha，将 karma-sinon-chai 加载到 Mocha 之中并运行当前页面加载的单元测试代码，最后将测试的结果输出到终端。

Karma 在我们引入 TDD 方法开发 Vue 组件后，它将是一个运行频次很高的工具。如果感觉运行速度慢的话，可以将 test/unit/karma.conf.js 内的代码覆盖性报告插件(coverage)暂时删除掉，因为生成这份报告并不是每次运行测试都必需的，更何况它是一个慢速插件。具体做法如下：

```
config.set({
  // ... 省略
  reporters: ['spec']
})
```

了解完 Karma 运行原理和作用，接下来我们了解一下每个工具使用的方法和具体完成的任务。

PhantomJS

PhantomJS (<http://phantomjs.org/>) 是一个无界面的、可脚本编程的 WebKit 浏览器引擎。它原生支持多种 Web 标准：DOM 操作、CSS 选择器、JSON、Canvas 以及 SVG。

一般来说，在我们的 Vue 单元测试代码中很少会直接以编码方式调用 PhantomJS 的功能，更多是利用 PhantomJS 具有高速的运行速度这一特点，优化每一次单元测试的效能，节省等待浏览器启动的漫长等待时间。

Karma 会自动加载 karma-phantomjs-lanucher 来引导 PhantomJS 启动，我们甚至不需要改动 karma.conf.js 内的任何配置。使用 PhantomJS 会比在 Karma 中使用 Chrome 作为宿主主要快上好几倍的启动时间。

MochaJS

Mocha (<http://mochajs.org/>) 是一个 JavaScript 测试框架，可以用来运行测试代码，它没有内置的 Assertion、Mock 和 Stub 功能。一般我们用 Chai 来为它提供断言，用 Sinon 为它提供 Mock 和 Stub 功能。Mocha 可以用来测试 Node.js 和浏览器的 JavaScript 代码。

Mocha 与 Jasmine 的语法非常相似, Mocha 配合 Sinon 可以更好地支持后端服务模拟的能力和异步测试调用, 这一点比 Jasmine 做得更优秀一些。我会在第 5 章再详细讲述它的用法。

执行单元测试的命令如下:

```
$ npm run unit
```

在单元测试环境内 Mocha 起到了两个作用, 首先它提供了单元测试框架与编写单元测试的规则, 如果你是一个 Ruby 开发者而且使用过 RSpec 的话, 你对此一定不会陌生:

```
describe('UkButton', () => {  
  it('应该输出 uikit 按钮的 HTML 结构', () => {  
    // ... 具体测试代码  
  })  
})
```

其次就是加载运行这些单元测试代码的解释运行器。

Chai

Chai 是一个提供 BDD 风格的代码断言库, 由于 Mocha 的代码断言非常简单, Chai 用于弥补 Mocha 的这一缺陷。例如:

```
expect(vm.$el.querySelectorAll('ul')).to.have.lengthOf(2)
```

expect 和 should 是 BDD 风格的, 二者使用相同的链式语言来组织断言, 但不同之处在于它们初始化断言的方式: expect 使用构造函数来创建断言对象实例, 而 should 通过为 Object.prototype 新增方法来实现断言(所以 should 不支持 IE); expect 直接指向 chai.expect, 而 should 则是 chai.should()。

我们在 vue-cli webpack 模块建立的 Vue 工程内编写单元测试是不需要手工配置 Chai 的, 因为 Chai 和 Sinon 被 Karma 通过 karma-sinon-chai 插件直接嵌入到单元测试的上下文中, 所以不需要 import 就能直接使用。

更多关于 Chai 的断言的资料请参考本书的“附录 A”。

Sinon

当测试的某个方法中, 需要去某个接口发送 HTTP 请求以获得数据, 如果你真实地发送某个请求, 那么当有一天你请求的这个服务器挂掉的时候, 你的单元测试就怎么也跑不过了。其实在测试的时候, 我们并不是真的关心这个接口是否存在(甚至是否实现), 我们需要模拟一个这样的接口来返回假的数据, sinonjs 就是解决这类问题的一个辅助库, 换个

专业的说法，Sinon 就是负责仿真的。

它主要提供方法调用侦测 (Spy)、接口仿真 (Stub) 和对象仿真 (Mock) 这三个方面的辅助功能。另外，vue-cli 的 webpack 模板所生成的单元测试环境采用 sinon-chai 这个联合库，它基于 Sinon 和 Chai 两个库直接提供了一套更方便使用代码的断言库，而这些配置早就被脚手架 vue-cli 配置好在项目里面等我们了。关于 Mocha、Sinon 和 Chai 的具体应用将在“Vue 的测试与调试技术”一章中通过具体示例一一讲述。

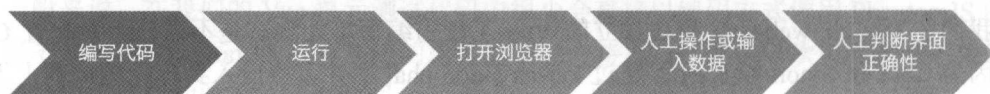
2.5 基于 Nightwatch 的端到端测试环境

不同公司和组织之间的测试效率迥异。在这个富交互和响应式处理随处可见的时代，很多组织都使用敏捷的方式来开发应用，因此测试自动化也成为软件项目的必备部分。测试自动化意味着使用软件工具来反复运行项目中的测试，并为回归测试提供反馈。

端到端测试又简称 E2E (End-To-End test) 测试，它不同于单元测试侧重于检验函数的输出结果，端到端测试将尽可能从用户的视角，对真实系统的访问行为进行仿真。对于 Web 应用来说，这意味着需要打开浏览器、加载页面、运行 JavaScript，以及进行与 DOM 交互等操作。简言之，单元测试的功能只能确保单个组件的质量，无法测试具体的业务流程是否运作正常，而 E2E 却正好与之相反，它是一个更高层次的面对组件与组件之间、用户与真实环境之间的一种**集成性测试**。

E2E 测试的意义在于可以通过程序固化和仿真用户操作，对于开发人员而言，基于 E2E 测试能极大地提高 Web 的开发效能，节约开发时间。

先来看看如果没有 E2E 测试下的一次从开发到手工测试成功的过程：



这个过程还属于简化过的，还没有包括在观察结果时要打开浏览器的调试窗口观看某些内部的运行变量或者网页代码结构。整个过程都是纯人工操作，人工操作最大的问题是一个程序可能要调试好几次，同样的操作就要重复数遍。即使有严格的规定，程序员们大多都还是随便地做“通过”式操作，尤其在输入样本数据时，绝大多数的程序员几乎都是乱输，出现得最多的就是各种随意的数字或者是“aaa”、“asd”、“aws”这样毫无意义的字符。以这种方式开发出来的程序在验收时产品经理或者客户会经常说一句话：“我上次试过是没有问题的！”这样的失误归根结底不在程序员本身，因为这是一种人性！一个人如果重复多次自己都觉得毫无意义的动作时，要不就逃避不做，如果不能逃避就会消极对待。

所以我们应该用更高效、更能弥补人性化缺陷和更有意义的办法来处理，这就是 E2E 测试，先来看看如果使用 E2E 测试后的开发过程将会变成什么：



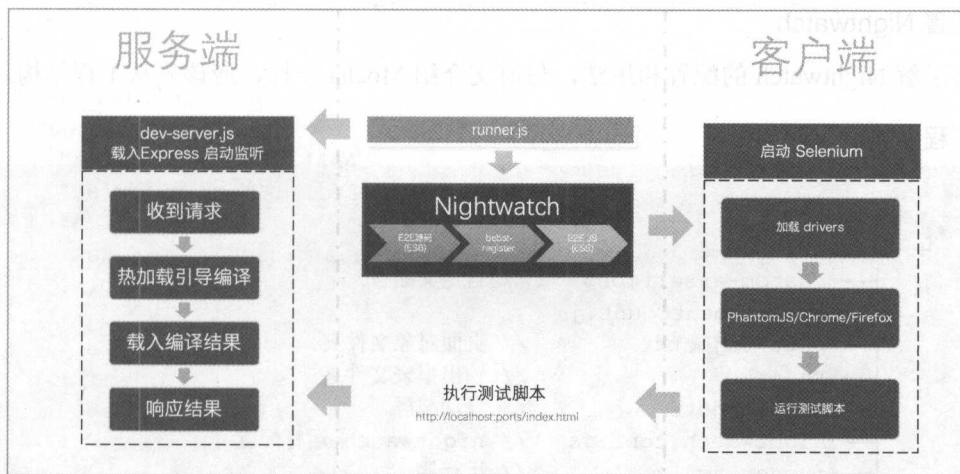
从运行测试开始，所有的一切都是自动的！这就是最大的区别，还有更重要的一点是，当我们要写出 E2E 测试时就需要对操作需求有深刻的理解，在这一过程中还有很大的机会对用户的操作进行优化，从而提高用户体验。

Nightwatch

vue-cli 的 webpack 模板也为我们准备了一个当下很流行的 E2E 测试框架——Nightwatch。

Nightwatch 是一套新近问世的基于 Node.js 的验收测试框架，使用 Selenium WebDriver API 以将 Web 应用测试自动化。它提供了简单的语法，支持使用 JavaScript 和 CSS 选择器来编写运行在 Selenium 服务器上的端到端测试。

这个框架在配置好后的具体工作流程如下图所示。



Nightwatch 采用 Fluent interface 模式 (https://en.wikipedia.org/wiki/Fluent_interface) 来简化端到端测试的编写，语法非常简洁易懂，正如以下代码所示。

```

this.demoTestGoogle = function (browser) {
  browser
    .url('http://www.google.com')
  }

```

```
.waitForElementVisible('body', 1000)
.setValue('input[type=text]', 'nightwatch')
.waitForElementVisible('button[name=btnG]', 1000)
.click('button[name=btnG]')
.pause(1000)
.assert.containsText('#main', 'The Night Watch')
.end();
}
```

我们可以从 Nightwatch 网站找到当前提供特性的列表：

- 简单但强大的语法。只需要使用 JavaScript 和 CSS 选择器，开发者就能够非常迅速地撰写测试。开发者也不必初始化其他对象和类，只需要编写测试规范即可。
- 内建命令行测试运行器，允许开发者同时运行全部测试——分组或单个运行。
- 自动管理 Selenium 服务器；如果 Selenium 运行在另一台机器上，那么也可以禁用此特性。
- 支持持续集成：内建 JUnit XML 报表，因此开发者可以在构建过程中，将自己的测试与系统（例如 Hudson 或 Teamcity 等）集成。
- 使用 CSS 选择器或 Xpath，定位并验证页面中的元素或是执行命令。
- 易于扩展，便于开发者根据需要，实现与自己应用相关的命令。

配置 Nightwatch

要了解 Nightwatch 的配置和用法，与前文介绍 Mocha 一样，应该先从工程结构入手。

工程结构

```
.
├── test
│   └── e2e
│       ├── custom-assertions // 自定义断言
│       │   └── elementCount.js
│       ├── page-objects // 页面对象文件夹
│       ├── reports // 输出报表文件夹
│       ├── screenshots // 自动截屏
│       ├── nightwatch.conf.js // nightwatch 运行配置
│       ├── runner.js // 运行器
│       └── specs // 测试文件
│           └── test.spec.js
```

以上是 vue-cli 为我们自动创建的 Nightwatch 工程结构，specs 是测试文件存放的文件夹，nightwatch.conf.js 是 Nightwatch 的运行配置文件。其他的目录将会在具体的章节逐一地进行讲述。

基本配置

Nightwatch 的配置项都集中在 `nightwatch.conf.js` 中, 其实这个配置也可以是一个 JSON 格式, 采用 JSON 格式只需要简单地对配置项写入一些常量即可。但使用模块的方式进行配置可以执行一些额外的配置代码, 这样则显得更为灵活。以下是我调整过的 `nightwatch.conf.js` 文件内容:

```
require('babel-register');
var config = require('.././config');
var seleniumServer = require('selenium-server');
var phantomjs = require('phantomjs-prebuilt');

module.exports = {
  "src_folders": ["test/e2e/specs"],
  "output_folder": "test/e2e/reports",
  "custom_assertions_path": ["test/e2e/custom-assertions"],
  "page_objects_path": "test/e2e/page-objects",
  "selenium": {
    "start_process": true,
    "server_path": seleniumServer.path,
    "port": 4444,
    "cli_args": {
      "webdriver.chrome.driver": require('chromedriver').path
    }
  },
  "test_settings": {
    "default": {
      "selenium_port": 4444,
      "selenium_host": "localhost",
      "silent": true,
      "launch_url": "http://localhost:" + (process.env.PORT || config.dev.port),
      "globals": {
      }
    },
    "chrome": {
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    },
    "firefox": {
      "desiredCapabilities": {
        "browserName": "firefox",
        "javascriptEnabled": true,
```

```

    "acceptSslCerts": true
  }
}
}
}

```

Nightwatch 的配置分为以下三类：

- 基本配置；
- Selenium 配置；
- 测试环境配置。

在配置模块中的所有根元素配置项都属于基本配置，用于控制 Nightwatch 的全局性运行的需要。下表为 Nightwatch 的基本配置项的详细说明。

配置项	类型	默认值	说明
src_folders	string / array	none	测试文件的存放目录
output_folder	string	tests_output	JUnit XML 测试报表的存放目录
custom_commands_path	string / array	none	自定义命令文件存放目录
custom_assertions_path	string / array	none	自定义断言文件存放目录
page_objects_path	string / array	none	页面对象文件存放目录
globals_path	string	none	全局模块配置文件目录，第 5 章会有具体说明
selenium	object	-	Selenium 服务器的运行配置项（详见下文）
test_settings	object	-	测试配置（详见下文）
live_output	boolean	false	配置是否在并行运行的情况下缓冲输出
disable_colors	boolean	false	在输出报表中禁用带有颜色的文字
parallel_process_delay	integer	10	指定并行模式下子进程启动的延时值（单位为毫秒）
test_workers	boolean/object	false	配置并行式运行测试
test_runner	string / object	"default"	声明其他的测试运行器（详见下文）

Selenium 配置

Selenium 是一组软件工具集，每一个工具都有不同的方法来支持测试自动化。大多数使用 Selenium 的 QA 工程师只关注一两个最能满足他们项目需求的工具。然而，学习所有的工具你将有更多选择来解决不同类型的测试自动化问题。这一整套工具具备丰富的测试功能，很好地契合了测试各种类型的网站应用的需要。这些操作非常灵活，有多种选择来定位 UI 元素，同时将预期的测试结果和实际的行为进行比较。Selenium 一个最关键的特性是支持在多浏览器平台上进行测试。

Selenium 诞生于 2004 年，当在 ThoughtWorks 工作的 Jason Huggins 在测试一个内部应

用时，作为一个聪明的家伙，他意识到相对于每次改动都需要手工进行测试，他的时间应该用得更有价值。他开发了一个可以驱动页面进行交互的 JavaScript 库，能让多浏览器自动返回测试结果。那个库最终变成了 Selenium 的核心，它是 Selenium RC（远程控制）和 Selenium IDE 所有功能的基础。Selenium RC 是开拓性的，因为没有其他产品能让你使用自己喜欢的语言来控制浏览器。

Selenium 是一个庞大的工具，所以它也有自己的缺点。由于它使用了基于 JavaScript 的自动化引擎，而浏览器对 JavaScript 又有很多安全限制，有些事情就难以实现。更糟糕的是，网站应用正变得越来越强大，它们使用了新浏览器提供的各种特性，都使得这些限制让人痛苦不堪。在 2006 年，一名 Google 的工程师 Simon Stewart 开始基于这个项目进行开发，这个项目被命名为 WebDriver。此时，Google 早已是 Selenium 的重度用户，但是测试工程师们不得不绕过它的限制。Simon 需要一款能通过浏览器和操作系统的本地方法直接和浏览器进行通话的测试工具，来解决 JavaScript 环境沙箱的问题。WebDriver 项目的目标就是要解决 Selenium 的痛点。

Selenium 1（又叫 Selenium RC 或 Remote Control）在很长一段时间内，Selenium RC 都是最主要的 Selenium 项目，直到 WebDriver 和 Selenium 合并而产生了最新且最强大的 Selenium 2。Selenium 1 仍然被活跃地支持着（更多是维护），并且提供一些 Selenium 2 短时间内可能不会支持的特性，包括对多种语言的支持（Java、JavaScript、Ruby、PHP、Python、Perl 和 C#）和对大多数浏览器的支持。

Selenium 2（又叫 Selenium WebDriver）代表了这个项目未来的方向，也是最新被添加到 Selenium 工具集中的。这个全新的自动化工具提供了很多了不起的特性，包括更内聚和面向对象的 API，并且解决了旧版本限制。Selenium 和 WebDriver 的作者都赞同两者各具优势，而两者的合并使得这个自动化工具更加强健。Selenium 2.0 正是于此的产品。它支持 WebDriver API 及其底层技术，同时也在 WebDriver API 底下通过 Selenium 1 技术为移植测试代码提供极大的灵活性。此外，为了向后兼容，Selenium 2 仍然使用 Selenium 1 的 Selenium RC 接口。

你可以到 <http://selenium-release.storage.googleapis.com/index.html> 下载 Selenium 的各个稳定版本。

在 Vue 项目中如果使用 vue-cli，那么 Nightwatch 将不需要进行任何的附加配置，否则你需要在命令行内安装 Selenium 的包装类库：

```
$ npm i selenium-server -D
```

Nightwatch 能引导 Selenium 的启动，实际上我们并没有必要去修改 Selenium 服务器的默认运行配置，在 nightwatch.conf.js 配置文件中只需要声明 Selenium 服务器的二进制执行

文件的具体路径即可，这个可以从 `selenium-server` 包提供的 `Selenium` 包装对象的 `path` 属性中获取，而无须将本机的物理路径写死到配置文件内。

```
var seleniumServer = require('selenium-server');

module.exports = {
  "selenium": {
    "start_process": true,
    "server_path": seleniumServer.path,
    "port": 4444,
    "cli_args": {
      "webdriver.chrome.driver": require('chromedriver').path
    }
  },
  // ... 省略
}
```

以下是 Selenium 的详细配置项说明：

配置项	类型	默认值	说明
start_process	boolean	false	配置是否自动管理 Selenium 进程
start_session	boolean	true	配置是否自动启用 Selenium 会话。当不需要与 Selenium 进行交互时可设置为 false
server_path	string	none	指定 Selenium jar 运行文件目录
log_path	string	none	Selenium 日志输出目录（默认为当前目录）
port	integer	4444	Selenium 服务器启动时占用的端口
cli_args	object	none	Selenium 命令行参数列表（详见下文）

cli_args 的配置

- `webdriver.firefox.profile`: Selenium 默认为每个会话创建一个独立的 Firefox 配置方案。如果你希望使用新的驱动配置可以在此进行声明。
- `webdriver.chrome.driver`: Nightwatch 同样可以使用 Chrome 浏览器加载测试，当然你要先下载一个 `ChromeDriver` 的二进制运行库对此进行支持。此配置项用于指明 `ChromeDriver` 的安装位置。除此之外，还需要在 `test_settings` 配置内使用 `desiredCapabilities` 对象为 Chrome 建立配置方案。
- `webdriver.ie.driver`: Nightwatch 也支持 IE，其作用与用法与 Chrome 相同，此处则不过多赘述。

测试环境配置

`test_settings` 内的项目将应用于所有的测试实例，在 E2E 测试中我们可以通过 Nightwatch 提供的默认实例对象 `browser` 获取这些配置值，`vue-cli` 为我们创建了 `default`、`firefox` 和 `chrome` 三个环境配置项，`default` 配置是应用于所有环境的基础配置选项，其他的配置项会自动覆盖与 `default` 相同的配置值。

`firefox` 和 `chrome` 这两个配置项是对两种浏览器的驱动进行描述和配置。对于其他语言或框架而言它们也是常客，但由于性能太低，在实战中通常只是个摆设，下文中我将会介绍一种实战效率更高的无头浏览器 PhantomJS，对其取而代之。

不要被 `vue-cli` 创建默认配置所迷惑，`test_settings` 并不单单只是对浏览器的一些基本运行参数的配置，它正确的用法是对 E2E 测试环境的配置。单元测试只能运行于开发环境内，而 E2E 却可以运行于本地环境与网络环境，更准确地说是开发环境与生产环境。所以这个配置项可以用以下的方式进行设置：

```
"test_settings": {
  "default": {
    "selenium_port": 4444,
    "selenium_host": "localhost",
    "silent": true,
    "launch_url": "http://localhost:" + (process.env.PORT || config.dev.port),
    "globals": {}
  },
  "dev": {
    "desiredCapabilities": {
      "browserName": "chrome",
      "javascriptEnabled": true,
      "acceptSslCerts": true
    }
  },
  "production": {
    "launch_url": "http://www.your-domain.com"
    "desiredCapabilities": {
      "browserName": "firefox",
      "javascriptEnabled": true,
      "acceptSslCerts": true
    }
  }
}
```

虽然与原有的配置只是在用词上做了一点点改变，但用词的改变将会彻底地改变我们对其的认知与思路！

下表是测试环境配置项的详细说明：

配置项	类 型	默 认 值	说 明
launch_url	string	none	启动测试的 URL。可以从测试的 browser.launchUrl 中获取它
selenium_host	string	localhost	配置正在接受连接的 Selenium 服务器主机地址
selenium_port	integer	4444	配置正在接受连接的 Selenium 服务器侦听的端口
request_timeout_options	object	60000	定义向 Selenium 服务器发出请求的超时选项
silent	boolean	true	是否显示扩展的 Selenium 命令日志
output	boolean	true	是否将结果输出到终端
disable_colors	boolean	false	禁止终端输出的结果文字采用任何的颜色
screenshots	object	none	设置 Selenium 生成的自动截屏的选项
username	string	none	设置 Authorization 头中用户名信息的安全票据。它可以从环境变量中读取，例如 "username" : "\${SAUCE_USERNAME}"
access_key	string	none	与 username 信息一同写入 Authorization 安全票据中的密码。同样可以从环境变量中读取："access_key" : "\${SAUCE_ACCESS_KEY}"
proxy	string	none	访问 Selenium 服务器时采用的代理服务器地址，例如，http://user:pass@host:port
desiredCapabilities	object	-	用于描述浏览器的相关参数并传递至 Selenium WebDriver，Selenium 将会在新的会话内启用此配置所指定的浏览器
globals	object	-	配置测试环境使用的全局变量或全局配置项
exclude	array	-	配置不进行测试的文件（可支持*通配符）
log_screenshot_data	boolean	false	当启用截屏时不显示 Base64 格式的图片数据
use_xpath	boolean	false	使用 XPath 作为测试的元素定位默认策略
cli_args	object	none	相当于 Selenium 配置中的 cli_args 项，可用于覆盖全局配置
end_session_on_fail	boolean	true	当任意测试单元失败时终止会话
skip_testcases_on_fail	boolean	true	当任意测试单元失败时忽略剩余的测试单元
output_folder	string/boolean	-	指定 JUnit XML 测试报表的输出目录
persist_globals	boolean	false	将此选项设置为 true 时，所有的测试单元将会共享相同的全局配置对象
compatible_testcase_support	boolean	false	适用于单元测试。当设置为 true 时，将允许测试写入与 Mocha 框架可互换的标准 Exports 接口
detailed_output	boolean	true	如果设置为 true 将会禁止断言输出具体的错误信息，而只显示测试单元是成功还是失败，此选项对并行式测试尤为有用

执行 E2E 测试

vue-cli 已经在 package.json 中配置了运行测试的指令：

```
$ npm run e2e
```

这个指令是默认启用 Chrome 运行环境的，如果指定运行环境可使用--env 选项：

```
$ npm run e2e --env
```

使用无头浏览器 PhantomJS

vue-cli webpack 脚手架模板非常好用，它将环境的复杂性降低了很多，但是却没有很好地诠释它里面采用的每个模块的理由和功能，以及它们的使用特点。这对于入门者来说确实是将门槛降到最低点，但从工程化开发的角度来说，只知道有这些环境或者工具的存在是远远不够的，在 Nightwatch 中就埋了一个这样的坑。

我们的开发环境在配置 Mocha 和 Karma 时就已经安装了 PhantomJS，但如果你细读 Nightwatch 的默认配置会惊奇地发现根本没有采用 PhantomJS，只是配置了 Chrome 和 Firefox！问题何在？一个字：慢！

我曾用一台 2013 年版标准配置（i5CPU、8GB 内存、1TB HDD 硬盘）的 iMac 跑本书下一章中的示例程序，运行一次的实际时间是 15 秒左右！仅仅一次就得 15 秒，那可以想象我们开发一个场景最少要做多少次的运行？Chrome 的启动是很慢的，我们做 E2E 这种自动化测试如果用真实浏览器的话只能将性能拖下来，生命不能耗费在毫无意义的等待中！所以我们才会选择 PhantomJS！没有默认配置 PhantomJS 作为主浏览器是这个环境的最大败笔。

办法总比问题多，所以如果没有，我们还可以自己动手来配置，其实方法也很简单。打开 nightwatch.conf.js，在 test_settings 配置段的下方加入以下内容：

```
"test_settings": {  
  "default": {  
    // ...  
  },  
  
  "phantom": {  
    "desiredCapabilities": {  
      "browserName": "phantomjs",  
      "javascriptEnabled": true,
```

```
    "acceptSslCerts": true,  
    "phantomjs.page.settings.userAgent": "Mozilla/5.0 (Macintosh; Intel Mac  
OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80  
Safari/537.36",  
  
    "phantomjs.binary.path": "node_modules/phantomjs-prebuilt/bin/phantomjs"  
  }  
}
```

Nightwatch 是通过 Selenium 加载一个 GhostDriver 来引导 PhantomJS 浏览器的，上面的内容就相当于告诉 Selenium 加载一个 GhostDriver，可执行程序则指向 npm 上安装的 phantomjs-prebuilt 包，再通过这个包来引导安装在本机上的 PhantomJS 启动。

按上文这样来引用 PhantomJS 的二进制程序的地址非常难看，还有原生配置中的 Selenium 执行程序地址也是一样的，这里介绍一个更 DRY 的方法来处理这些路径：

```
var seleniumServer = require('selenium-server');  
var phantomjs = require('phantomjs-prebuilt');
```

```
module.exports = {  
  // ...省略
```

```
  "selenium": {  
    // ... 省略
```

```
    "server_path": seleniumServer.path,  
  },
```

```
  "test_settings": {  
    // ... 省略
```

```
    "phantom": {  
      "desiredCapabilities": {  
        // ... 省略
```

```
        "phantomjs.binary.path": phantomjs.path  
      }  
    }  
  }  
}
```

```
// ... 省略
```

```
}
}
```

做完这个简单的优化后就可以打开 `runner.js` 文件找到:

```
if (opts.indexOf('--env') === -1) {
  opts = opts.concat(['--env', 'chrome'])
}
```

将 `chrome` 改为 `phantom` 就行了:

```
if (opts.indexOf('--env') === -1) {
  opts = opts.concat(['--env', 'phantom'])
}
```

重新加载测试程序, 在同一台 iMac 上的运行速度直接降到了 5 秒, 测试运行速度提升了 3 倍! 如果你有配置更好的机器, 将硬盘换成 SSD 之后会有更惊人的速度。

Nightwatch 与 Cucumber

如果你正在开发的项目的业务复杂性不大, 可以直接使用 Nightwatch 推荐的链式调用写法。但是当这种做法真正应用在业务流程较多, 或者业务操作相对复杂的应用场景时, 你会觉得总有写不完的 E2E 测试, 因为这么做 E2E 测试是没有办法一次性覆盖所有需求的!

E2E 测试其实是行为式驱动开发的实现手法, 如果跳过了行为式驱动开发的分析部分直接编写 E2E, 其结果只能是写出一堆严重碎片化的测试场景, 甚至会出现很多根本不应该出现的操作。

幸好 Nightwatch 具有很好的扩展性与兼容性, 能集成最正统的 BDD 测试框架 Cucumber (<https://cucumber.io/>)。Cucumber 是原生于 Ruby 世界的 BDD 框架, 但它也有很多的语言实现版本, 我们可以安装一套专门为 Nightwatch 编写的 Cucumber 版本——nightwatch-cucumber (<https://github.com/mucsi96/nightwatch-cucumber>)。本章只介绍关于环境与工具的配置, 而关于如何来应用 BDD, 内容已经超出了本书的知识范围, 如果有兴趣的话可以参考《攀登架构之巅》一书中行为式驱动开发的章节内容。

```
$ npm i nightwatch-cucumber -D
```

然后在 `~/.test/e2e/nightwatch.conf.js` 文件中加入对 Cucumber 的配置:

```
// ... 省略
require('babel-register');

require('nightwatch-cucumber')({
```



```
nightwatchClientAsParameter: true,  
featureFiles: ['test/e2e/features'],  
stepDefinitions: ['test/e2e/features/step_definitions'],  
jsonReport: 'test/e2e/reports/cucumber.json',  
htmlReport: 'test/e2e/reports/cucumber.html',  
openReport: false  
});
```


第3章 路由与页面间导航

真实的工程项目并不会像开篇举出的示例仅需要一个页面就能完成，一个完整的业务系统或者网站平台项目要编写的页面往往是几十个甚至上百个，所以当建立工程化的项目结构后，摆在面前的问题就是：

- (1) 项目中应该有多少个页面？
- (2) 页面与页面之间存在何种关系，应该如何进行导航？
- (3) 哪里是程序的入口？应该先从哪个页面开始入手？

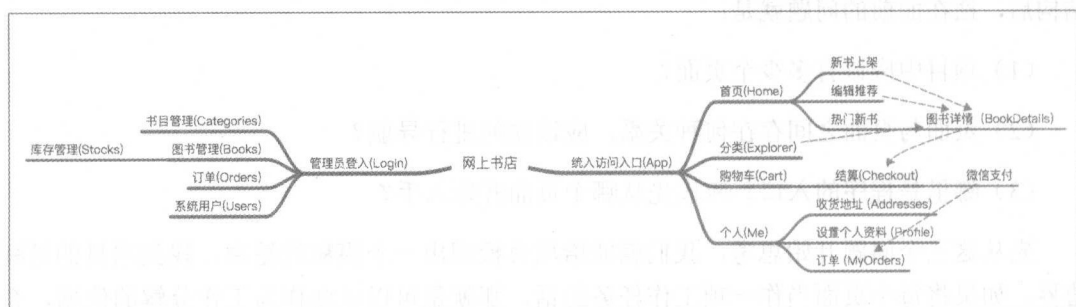
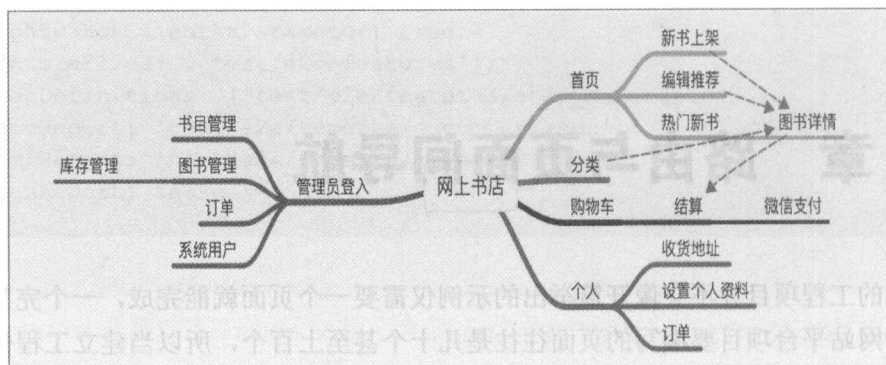
先从这三个问题开始思考，我们就能给项目梳理出一个模糊的轮廓，找到项目的可视边界。如果将每个页面当作一项工作任务的话，那就是可以以此作为工作分解的依据，合理地分配人员与安排时间。

从思维导图到网站地图

本章将展示一个微信网上书店的示例，分析接到这样的项目时我是如何通过思维导图辅助来进行思考和设计的，如何将这个思维导图进一步具象化、视觉化，再重新化为逻辑化的代码的。

我很喜欢用思维导图作为辅助我进行思考与设计的工具，如果经常使用它，你会发现一个小小的发散性的图形工具能很有效地将很多杂乱无章的问题梳理得井井有条。如果你还没有开始使用它，我推荐你可以看看思维导图之父的一本书《思维导图》（【英】东尼·博赞 巴利·博赞 著 卜煜婷译 2015 化学工业出版社）。你一定可以从那学到很多思考的方法。由于思维导图的勾画实在是太自由了，用于梳理一个网站的结构的话，我们只需要将每个页面视为一个节点，思维路径当作导航的路径，这样可以很快地得出下面这一张图。

这个思维导图很好地诠释了整个微信网上书店的逻辑结构，整个系统的功能一目了然。但如果以此作为页面间导航图，也就是俗称的网站地图的话就还有欠缺。我们来为每个节点给予一个英文的命名，并且将多个提供给前台用的访问入口进行合并，进一步梳理：



设计原型

有了清晰的网站地图后，我们就可以用工具将这个在大脑中模糊的界面真实地呈现出来。我见过很多开发人员随便拿张纸画一下，又或者找些其他的图形工具画个示意图就开始动手编码。这种做法的结果是，只有设计者本人做出来的程序有可能与想象的一致，而绝大多数情况下拿这样的草图给 10 个程序员就会有 10 种不同的实现！设计必须明确细致，界面设计将直接影响操作的行为，哪怕是简单的线条颜色都应该标明具体的色值。欠下的技术债务始终要偿还，在设计之初不细致就会在交付期偿还，这是多少前人从各种各样的失败中得到的不变铁律。

所以，我宁愿在设计图上多花一点心思，尤其是与用户交互的设计，尽力将图纸做到与真实交付的产品是一致的。我推荐使用 Sketch，这是一个非常实用的矢量图工具，对于开发人员来说极易上手，因为它就是为了设计软件原型图而生的。无论是从 Window、Web 到 App 的原型图，它可以确保我们能将原型图做到与真实的产品毫无差异的程度，以下就是本示例中原型的一部分截图。

有了网站地图和设计原型，接下来我们就开始正式进入 Vue，使用官方提供的路由库 `vue-router` 为我们的项目建立动态、完整的程序骨架。



3.1 vue-router

从传统意义上说，路由就是定义一系列的访问地址规则，路由引擎根据这些规则匹配并找到对应的处理页面，然后将请求转发给页进行处理。可以说所有的后端开发都是这样做的，而前端路由是不存在“请求”一说的。前端路由是直接找到与地址匹配的一个组件或对象并将其渲染出来。改变浏览器地址而不向服务器发出请求有两种做法，一是在地址中加入#以欺骗浏览器，地址的改变是由于正在进行页内导航；二是使用 HTML5 的 window.history 功能，使用 URL 的 Hash 来模拟一个完整的 URL。

Vue.js 官方提供了一套专用的路由工具库 vue-router。vue-router 的使用和配置都非常简单，而且代码清晰易读，很容易上手。

将单页程序分割为各自功能合理的组件或者页面，路由起到了一个非常重要作用。它就是连接单页程序中各页面之间的链条，除了在本章中会重点对其用法通过开发实例进行详细介绍，我还将其他一些关于路由的细小的运用方法分散在各个章节之中，既然它是“链条”，那么在每个环节都将出现它的身影。

安装

```
$ npm i vue-router -D
```

vue-router 实例是一个 Vue 的插件，我们需要在 Vue 的全局引用中通过 Vue.use() 将它接入到 Vue 实例中。在我们的工程中，main.js 是程序入口文件，所有的全局性配置都会

在这个文件中进行。

打开 main.js 文件并加入以下的引用：

```
import Vue from 'vue'
import VueRouter from 'vue-router'
Vue.use(VueRouter)
```

这样就完成了 vue-router 最基本的安装工作了。

路由配置

接下来就需要开始一项对整个项目来说都起到关键性意义的工作了，这就是路由表的定义，或者叫路由配置。

在开始之前，我们得先建立一些基本的概念，这样会更便于我们的设计与实现。

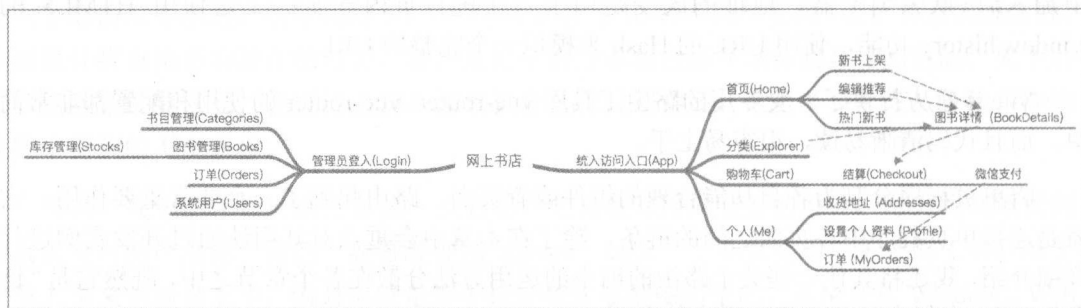
单页式应用是没有“页”的概念的，更准确地说，Vue.js 是没有页面这个概念的，Vue.js 的容器就只有组件。但我们用 vue-router 配合组件又会重新形成各种的“页”，那么我们可以这样来约定和理解：

(1) 页面是一个抽象的逻辑概念，用于划分功能场景。

(2) 组件是页面在 Vue 的具体实现方式。

一定要谨记以上这两点，因为在后面的内容中还会围绕这个约定对我们的项目进行结构性的优化。

这里的路由的定义方法就是将思维导图演化为页面导航图，在上一节中我们已经设计出了一份完整的网站地图：



我们先来实现网站地图右侧的功能，也就是面向最终用户的前台功能。按照我们的设计，统一程序入口应该就是一个带有分页导航栏的页面容器，用户打开我们的程序看到的第一个页面应该是“首页”，也称之为默认路由，而其他的页面（分类、购物车、个人）都是根页面，在 Tab 导航栏内的是顶层页面。按照上文的约定：页面就是组件，那么一个路由定义就该与一个组件相对应，具体应该如下表所示。

名 称	路 由	组 件
首页	/home	Home.vue
分类	/explorer	Explorer.vue
购物车	/cart	Cart.vue
我	/me	Me.vue

*.vue 文件是 Vue 的单页式组件文件格式，它可以同时包括模板定义、样式定义和组件模块定义。

首先，我们在项目目录下分别建立这四个顶层页面的 Vue 组件文件：

```
.
├── src
│   ├── App.vue
│   ├── assets
│   ├── Home.vue
│   ├── Explorer.vue
│   ├── Cart.vue
│   ├── Me.vue
│   └── main.js
└── webpack.config.js
```

这些新建的页面组件内容暂时都可以是同样的结构：

```
<!--/Home.vue-->
<template>
  <div>首页</div>
</template>
<style></style>
<script>
  export default {}
</script>
```

接下来就是在 main.js 文件中定义路由与这些组件的匹配规则了。VueRouter 的定义非常简单易懂，只需要创建一个 VueRouter 实例，将路由 path 指定到一个组件类型上就可以了，代码如下所示。

```
main.js
import Vue from 'vue'
import VueRouter from 'vue-router'
import App from './App.vue'

// 引入创建的四个页面
import Home from './Home.vue'
```



```
import Explorer from './Explorer.vue'
import Cart from './Cart.vue'
import Me from './Me.vue'

// 使用路由实例插件
Vue.use(VueRouter)

const router = new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    // 将页面组件与 path 指定的路由关联
    {path: '/home', component: Home},
    {path: '/explorer', component: Explorer},
    {path: '/cart', component: Cart},
    {path: '/me', component: Me}
  ]
})

new Vue({
  el: '#app',
  // 将路由实例添加到 Vue 实例中
  router,
  render: h => h(App)
})
```

在真实的项目开发中，由于我们可能需要引入很多不同的第三方库，如上述代码所示，每个库基本上都需要进行一些基本的配置，很明显这些配置会根据实际开发的需要不断地增加内容。如果我们将这些配置全部都放在 `main.js` 文件内，那么这个全局配置文件就会变得越来越长，非常不利于维护。所以我们可以将这些配置独立出来，正如现在讲述的路由配置，我们可以将它独立到一个 `routes.js` 文件内，那么 `main.js` 的内容就会被减缩成以下的样子：

```
import Vue from 'vue'
import App from './App.vue'
import router from './config/routes'

new Vue({
  el: '#app',
  // 将路由实例添加到 Vue 实例中
  router,
  render: h => h(App)
})
```

当我们增加页面或者路由时就直接修改 `routes.js` 的内容：

```

import Vue from 'vue'
import VueRouter from 'vue-router'

// 引入创建四个页面
import Home from './Home.vue'
import Explorer from './Explorer.vue'
import Cart from './Cart.vue'
import Me from './Me.vue'

// 使用路由实例插件
Vue.use(VueRouter)

export default new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    // 将页面组件与 path 指定的路由关联
    {path: '/home', component: Home},
    {path: '/explorer', component: Explorer},
    {path: '/cart', component: Cart},
    {path: '/me', component: Me}
  ]
});

```

同理，以后凡是遇到配置文件，就可以将其放置到 `config` 目录下，在 `main.js` 中仅引用配置到全局 `Vue` 实例内的配置对象，这样会非常便于我们维护各种各样的配置。

3.2 路由的模式

我们要了解一点，传统意义上路由是由多个 URL 或者 URL 规则组成的，对服务端而言，网页的访问是无状态的，称之为服务端路由。而浏览器的 History API 则给予了一种实现可状态化页面的可能，因为页面的跳转（URL 的改写）并不会出现页面刷新，这样一来状态就被维护在浏览器的 History 的内部状态存储之中。

前文中，我们创建 `VueRouter` 实例时用了 `mode:history` 的参数，这个值的意思是使用 history 模式，这种模式充分利用了 `history.pushState` API 来完成 URL 跳转而无须重新加载页面。

如果不使用 history 模式，当访问 home 的时候地址就会变为：

```
http://localhost/#home
```

反之为：

http://localhost/home

这就是 history 模式与 hash 模式的区别了，除了它们，还有一种叫 abstract 的模式，以下是这三种模式的详细解释。

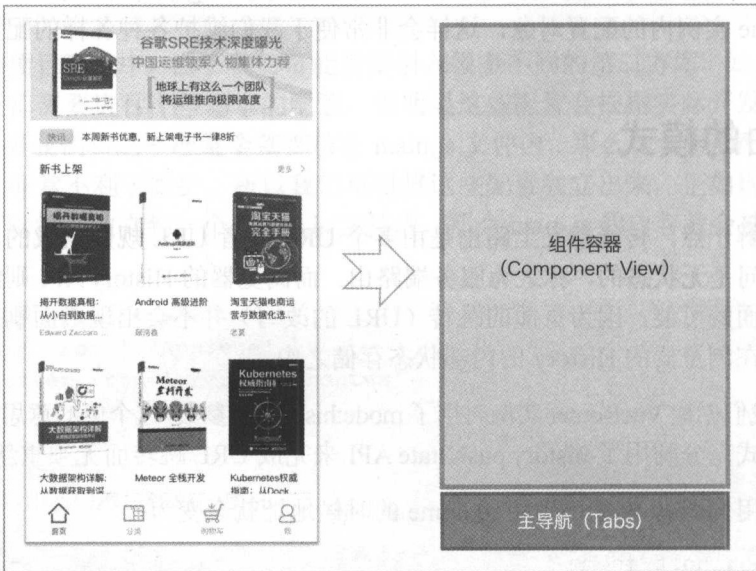
- Hash: 使用 URL hash 值来作为路由。支持所有浏览器，包括不支持 HTML5 History API 的浏览器。
- History: 依赖 HTML5 History API 和服务器配置。查看 HTML5 History 模式。
- Abstract: 支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式。

base 为应用的基路径。例如，整个单页应用服务在/app/下，那么 base 就应该设为“/app/”，当你在 HTML5history 模式下使用 base 选项之后，所有的 to 属性都不需要写基路径了。

3.3 路由与导航

我们已经在 routers.js 定义中建立了路由与组件之间的关系，接下来就要将这些路由关系在视图中互动起来。

按照网站地图，我们先来分析一下入口程序的网页结构，如下图所示。

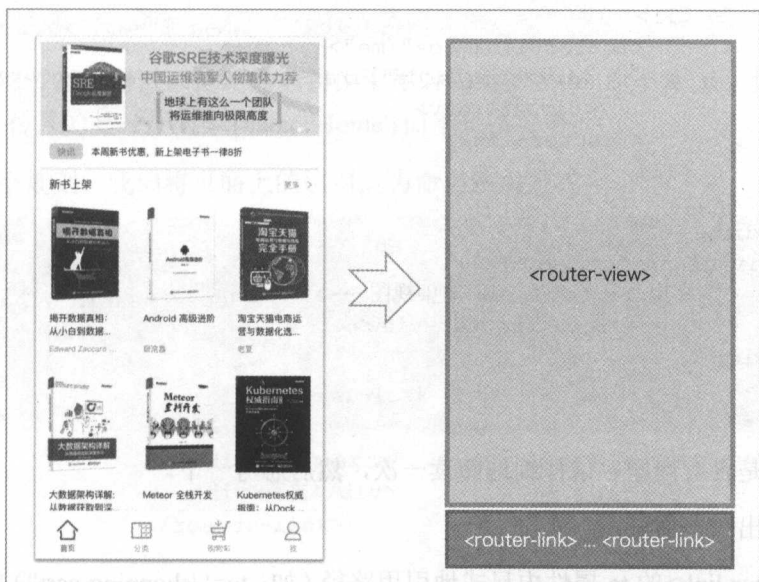


当我们点击下方的主导航区域内的图标时（切换路由），组件容器（Component View）就应该渲染相应的组件，其实这也就是 vue-router 所提供的功能。

vue-router 提供了两个指令标签（directive）组件来处理这个导航与自动渲染逻辑：

- `<router-view>`——渲染路径匹配到的视图组件，它还可以内嵌自己的`<router-view>`，根据嵌套路径渲染嵌套组件。
- `<router-link>`——支持用户在具有路由功能的应用中（点击）导航。

用这两个 directive 替换上述的设计图我们就能清楚理解它们在页面布局上的用法了：



经过这样的一番分析与准备，程序主入口 `App.vue` 组件的功能已基本清晰，接下来我们就在 `App.vue` 文件内用`<router-link>`和`<router-view>`改写`<template>`中的模板内容：

```
<template>
  <div id="app">
    <div class="tabs">
      <ul>
        <li>
          <!-- 使用 router-link 指向定义的 path -->
          <router-link to="/home">
            <div></div>
            <div>首页</div>
          </router-link>
        </li>
        <li>
          <router-link to="/categories">
            <div></div>
            <div>分类</div>
          </router-link>
        </li>
      </ul>
    </div>
    <router-view>
  </div>
</template>
```

```

        </router-link>
      </li>
      <li>
        <router-link to="/shopping-cart">
          <div></div>
          <div>购物车</div>
        </router-link>
      </li>
      <li>
        <router-link to="/me">
          <div></div>
          <div>我</div>
        </router-link>
      </li>
    </ul>
  </div>
  <div class="content">
    <!-- 用 router-view 渲染视图 -->
    <router-view></router-view>
  </div>
</div>
</template>

```

上述代码是否有问题？请仔细地阅读一次，然后思考一下。

我们来提出一些质疑：

- 在<router-link>的 to 属性内显式地引用路径（如：to="/shopping-cart"）的做法对吗？
- 这个路径不是在 VueRouter 的全局配置中已经出现过吗？
- 万一需要修改这个 URL 怎么办，在使用过的地方都要改一次？

无论我们是做服务端开发还是前端开发，路由的使用都有一个明确的原则：**不直接引用路由定义**。

这是一个很容易想到的问题，当显式引用路由定义的 URL 一旦产生变更，所有引用的地方都需要改，当程序开始规模化时路由变得越来越多的时候，这种变更所带来的工作量是可想而知的！所以我们应该从一开始就注意到这一点，vue-router 提供了一种隐式的路由引用方式，vue-router 将之称为“命名路由”，简单点说就是通过路由的名称引用取代 URL 的直接引用。为此我们需要先在 VueRouter 的配置上做一些重构：

```

const router = new VueRouter({
  mode: 'history',
  base: __dirname,
  routes: [
    // 将页面组件与 path 指定的路由关联

```



```

    {name:'Home', path: '/', component: Home},
    {name:'Categories',path: '/categories', component: Category},
    {name:'ShoppingCart',path: '/shopping-cart', component: ShoppingCart},
    {name:'Me',path: '/me', component: Me}
  ]
})

```

在<router-link>内通过名称引用路由需要向 to 属性传入一个对象显式声明路由的名称:

```
<router-link :to="{ name : 'Home' }">
```

有一个细节需要留意,使用命名路由引用时采用的是to 而不是 to,因为这个时候向<router-link>传入的是一个对象{name:'Home'}而不是字符串。

按照这个规则,我们将页面上的引用改为命名路由方式:

```

<template>
  <div id="app">
    <div class="tabs">
      <ul>
        <li>
          <!-- 使用 router-link 指向定义的 path -->
          <router-link :to="{ name : 'Home' }">
            <div></div>
            <div>首页</div>
          </router-link>
        </li>
        <li>
          <router-link :to="{ name: 'Categories' }" >
            <div></div>
            <div>分类</div>
          </router-link>
        </li>
        <li>
          <router-link :to="{ name: 'ShoppingCart' }">
            <div></div>
            <div>购物车</div>
          </router-link>
        </li>
        <li>
          <router-link :to="{ name: 'Me' }">
            <div></div>
            <div>我</div>
          </router-link>
        </li>
      </ul>
    </div>
  </div>
</template>

```

```

    </div>
    <div class="content">
      <!-- 用 router-view 渲染视图 -->
      <router-view></router-view>
    </div>
  </div>
</template>

```

切记：虽然命名路由的方式会比直接引用 path 多写一些代码，但这是值得的，因为一旦遇到路径的修改只需要在 main.js 的全局路由设置中进行修改而不用在每个用到的地方都改一次！

输出指定元素

<router-link>组件支持用户在具有路由功能的应用中（点击）导航。通过 to 属性指定目标地址，默认渲染成带有正确链接的<a>标签。也就是说，上述代码最终的输出结构是这样的：

```

<ul>
  <li>
    <a href="#">
      <div></div>
      <div>首页</div>
    </a>
  </li>
  <li> ... </li>
</ul>

```

其实，我们并不需要输出<a>元素标记，因为我们并没有具体的链接地址，由元素同样可以处理来自用户的点击切换路由的事件。<router-link>可以通过配置 tag 属性生成别的标签，利用这个属性我们可以直接输出而节省更多的代码，将上述代码进行重构：

```

<router-link :to="{ name : 'Home' }"
  tag="li">
  <div>
    
  </div>
  <div>首页</div>
</router-link>

```

最终的输出结果就变成：

```

<ul>
  <li>
    <div></div>

```

```
<div>首页</div>
</li>
<li> ... </li>
</ul>
```

动态路由

现在我们已经建立了整个微信网上书城的入口结构以及顶层导航结构了，因为4个顶层页面还没有内容，所以由顶层页面作为导航入口进入的二级和三级页面还没有建立路由，它们是：

- 图书详情 (BookDetails);
- 结算 (Checkout);
- 收货地址 (Address);
- 个人资料 (Profile);
- 订单 (Orders)。

在正式的开发场景下，我们应该先实现这些顶层页面还是先实现所有的路由定义呢？答案很明显是后者，设计网站地图的原意就是能从一开始就清楚地知道我们的项目由多少的页面组成，哪些页面实现了，哪些还需要实现或者完善，尤其是当我们正在处于一个多人协作开发环境之中，这尤为重要，因为路由是连接整个项目的主线。接下来根据前文提供的方法，先按照以上列表的命名建立余下的各个页面文件，然后回到路由配置中来 (main.js)。

首先是图书详情，这是一个拥有多个链接入口的页面，同时它也要转换至结算页面（在界面设计中图书详情内可以点击“立即购买”，然后直接转跳至结算页），每个图书详情显示的是单本书的详细信息，这些数据是存在于服务端的。也就是说，当转到这个页面时它应该先从服务端读取这本书的具体数据对象，那这个读取动作也必然需要通过图书的唯一编号作为查询条件。可以得到一个结论：图书详情页需要从路由中读取一个书号的参数，然后以此书号查询服务器的图书数据对象。

将参数作路由的习惯性做法不是在路由后面以“参数=值”的方式，因为这种方式已经非常过时与老旧，而且可读性极差，我们可以将参数融入到路由的路径定义之内成为路径的一部分，使之更具有可读性，我们称这种参数为“动态路径参数”，具体的做法是在参数名之前加上“:”，然后将参数写在路由的 path 内，具体定义如下：

```
routes: [{
  name: 'BookDetails',
  path: '/books/:id'
  component: BookDetails
}]
```

这样定义以后，vue-router 就会自动匹配所有/books/1、/books/2、...、/books/n 形式的路由模式，因为这样定义的路由的数量是不确定的，所以也被称为“动态路由”。

在<router-link>中我们就可以加入一个 params 的属性来指定具体的参数值：

```
<router-link :to="{name:'BookDetails', params: { id: 1 }}">
  <!-- ... -->
</router-link>
```

如果同时要传递多个参数，只要按以上的命名方法来加入参数，传递时在 params 中对应地声明参数值即可，vue-router 只要匹配到路由模式的定义就会自动对参数进行分解取值。

那在图书详情页内又如何从路由中重新将这个:id 参数读取出来呢？做法非常简单，可以通过\$router.params 这个属性获取指定的参数值，例如：

```
export default {
  created () {
    const bookID = this.$router.params.id
  }
}
```

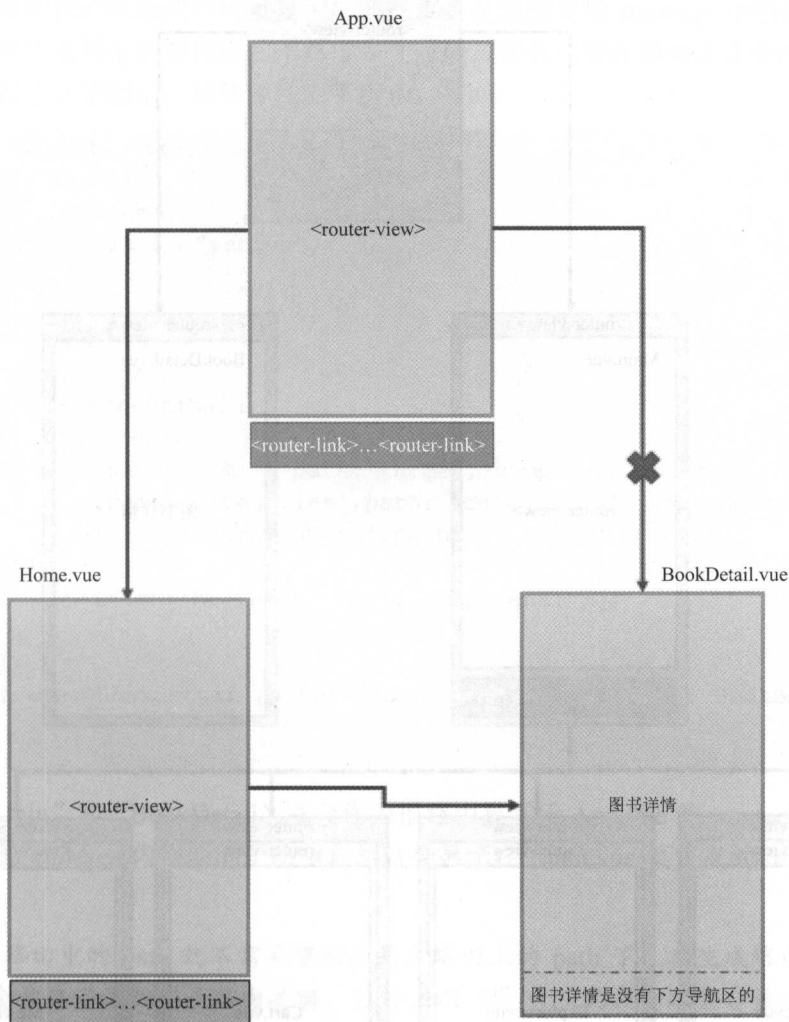
顺便提一下，当使用路由参数时，例如从/books/1 导航到 books/2，原来的组件实例会被复用。因为两个路由都渲染同一个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着组件的生命周期钩子不会再被调用，也就是说 created、mounted 等钩子函数在页面第二次加载时将失效。那么，当复用组件时，想对路由参数的变化做出响应的话，就需要在 watch 对象内添加对\$route 对象变化的跟踪函数：

```
export default {
  template: '...',
  watch: {
    '$route' (to, from) {
      // 对路由变化作出响应
    }
  }
}
```

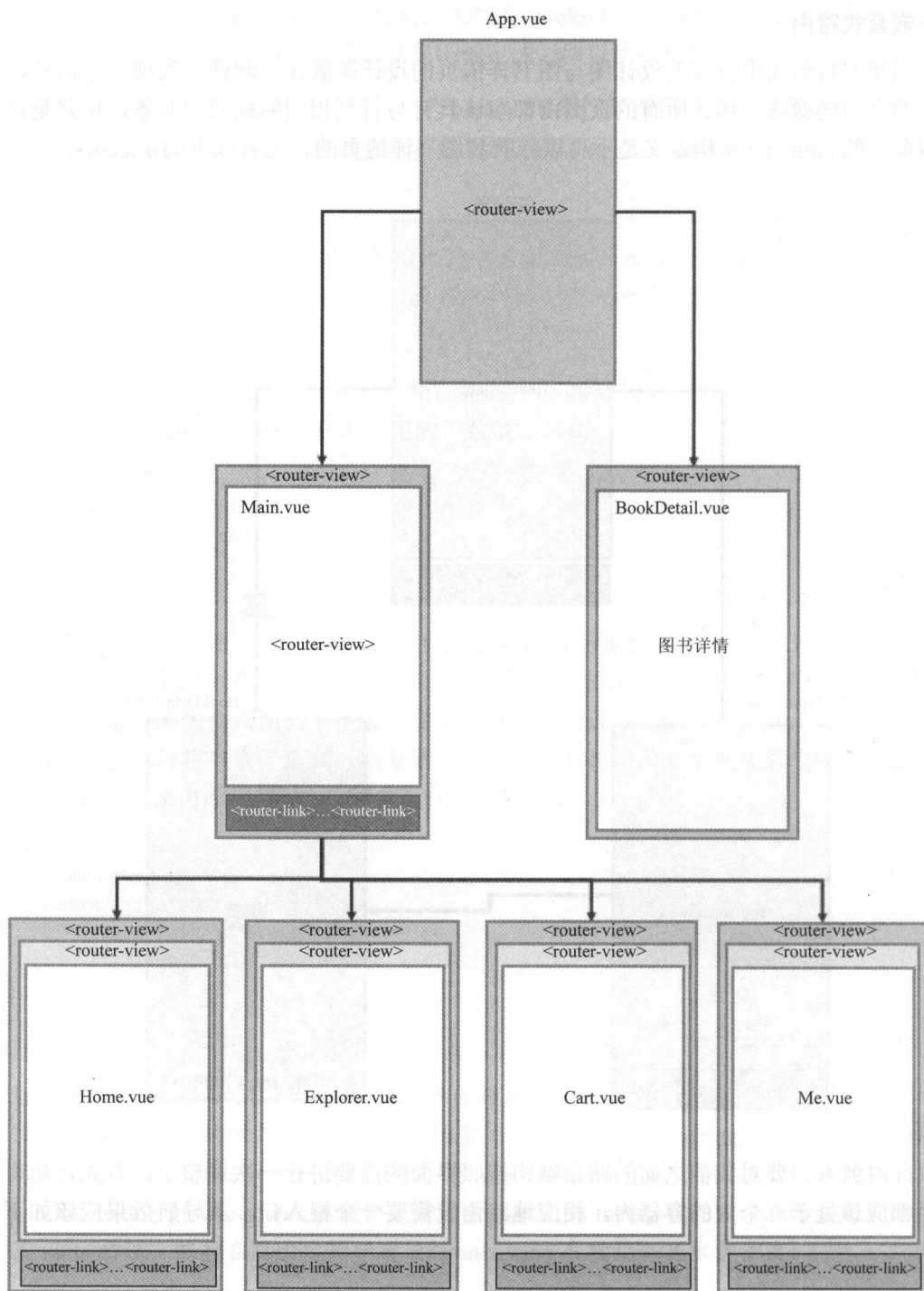
\$router.params 定义的参数必然是整个路由的其中一部分，vue-router 还可以让我们使用“/path?参数=值”的方式，也就是俗称的查询字符串（Query string）传递数据。如果要从\$router 中读取 Query string 的参数，可以使用\$router.query.参数名的方式读取。除了 params 和 query，vue-router 还提供一种常量参数定义 meta，我们可以在路由定义中先定义 meta 的值，然后在路由实例中通过\$router.meta 参数获取具体常量值。

嵌套式路由

当我们将前文中首页的设计图与图书详情页的设计图放在一起就会发现一个问题，如果按照之前的做法，那么所有的页面内都应该具有与首页相同的底部导航条，也就是说如果按前文的 App.vue 结构定义是不可以导航到图书详情页的，请看以下的示意图：



此时就有必要对我们之前的路由结构按照界面的需要进行一次调整了。首先，所有的页面都应该处于一个大的容器内，相应地路由就需要一个根入口，其导航效果应该如下图所示。



由上图可知, App.vue 页面除了<router-view>就不需要其他的元素了,说白了就是一个最大的页面容器。原有导航部分的内容应该移到一个新的页面上,也就是上图中的 Main.vue。Main.vue 中的<router-view>相对于 App.vue 中的<router-view>,就是一个用于显示子路由的视图。

关于视图的代码此处就不再重复了,现在重点是怎样重构 routes.js 中的路由配置,要将路由显示到子视图中就要相应的子路由与之对应,那么只要要在路由定义中用 children 数组属性就可以定义子路由,具体做法如下所示。

```
export default = new VueRouter({
  mode: 'history',
  base: __dirname,
  linkActiveClass: "active",
  routes: [
    {
      name: 'Main',
      path: '/',
      component: Main,
      children: [
        {name: 'Home', path: 'home', component: Home},
        {name: 'Categories', path: 'categories', component: Category},
        {name: 'ShoppingCart', path: 'shopping-cart', component:
ShoppingCart},
        {name: 'Me', path: 'me', component: Me}
      ]
    },
    {name: 'BookDetail', path: '/books/:id', component: BookDetail}
  ]
})
```

这样“Main”和“BookDetail”就会作为根路由显示于 App.vue 的<router-view>中,而在“Main”的 children 内声明的子路由自然就会显示在 Main.vue 页定义的<router-view>子视图中了。

注意:子路由中的 path 就不需要重新声明主路由上的 path 了,在生成路由时,主路由的 path 会被自动添加到子路由之前。另外,以“/”开头的嵌套路径会被当作根路径,所以不要在子路由上加上“/”。

切页动效

当我们做完上述所有的修改后,页面就能按我们之前设计的那样导航了。但在实际的使用过程中会发现页面之间的切换是非常生硬的,因为我们没有使用任何的动画效果。幸

好，在 Vue2 中使用 CSS 动画是一件非常简单的事，Vue 在插入、更新或者移除 DOM 时，提供了多种不同方式的应用过渡效果。

Vue 提供了 `transition` 的封装组件，在下列情形中，可以给任何元素和组件添加进入（entering）和退出（leaving）的过渡效果。在 Vue 官方的中文手册关于“过滤效果”（<https://cn.vuejs.org/v2/guide/transitions.html>）一文中有很多例子值得一读。它的应用非常简单，只要在 `<transition>` 组件内加入需要参与 CSS 过滤的元素，然后在 `name` 属性上声明使用的 CSS 类名，最后按照以下方式定义 CSS 伪类就可以启用过滤效果了。

有 4 个（CSS）类名在 `enter/leave` 的过渡中切换，以下是这 4 个类名的命名规则和作用。

- CSS 类名-`enter`：定义进入过渡的开始状态。在元素被插入时生效，在下一个帧移除。
- CSS 类名-`enter-active`：定义进入过渡的结束状态。在元素被插入时生效，在 `transition/animation` 完成之后移除。
- CSS 类名-`leave`：定义离开过渡的开始状态。在离开过渡被触发时生效，在下一个帧移除。
- CSS 类名-`leave-active`：定义离开过渡的结束状态。在离开过渡被触发时生效，在 `transition/animation` 完成之后移除。

根据以上规则我们为页面加上滑动淡出的过渡效果 `slide-fade`，具体代码如下所示。

```
<template>
  <transition name="slide-fade">
    <router-view>
  </router-view>
</transition>
</template>

<style>
.slide-fade-enter-active {
  transition: all .3s ease;
}

.slide-fade-leave-active {
  transition: all .3s cubic-bezier(1.0, 0.5, 0.8, 1.0);
}

.slide-fade-enter, .slide-fade-leave-active {
  transform: translateX(-430px);
  opacity: 0;
}
```

```
}
</style>
```

3.4 导航状态样式

我们回到 Home.vue 页，当用户点击 Tabs 上的任意一个标签组件时，组件应该进入一个“激活”的状态，显示为红色。这一点 VueRouter 也为我们想到了，在默认情况下当 `<router-link>` 对应的路由匹配成功时，就会自动设置 class 属性值为 `.router-link-active`，如果我们想要将“激活”状态样式类命名为 `active`，可以通过 `active-class` 属性进行设置，例如：

```
<router-link :to="{ name : 'Home' }"
  tag="li"
  active-class="active">
  <div>
    
  </div>
  <div>首页</div>
</router-link>
```

如果在页面上都是这样显式的声明，那么就需要在每个 `<router-link>` 组件元素上都要写一次，这样就不 DRY 了。我们还可以有另一个选择，就是在 VueRouter 的全局配置上进行声明，直接将 `.router-link-active` 这个默认值改为 `active`，在 `main.js` 文件内的 VueRouter 配置中加入以下的语句：

```
const router = new VueRouter({
  // ... 省略
  linkActiveClass: "active",
})
```

通过 `linkActiveClass` 全局属性就能进行统一的设置了。

精确匹配与包含匹配

`<router-link>` 添加“激活”状态样式类的默认依据是对 URL 地址的全包含匹配。举个例子，如果当前的路径是 `/home`，那么 `<router-link to="/">` 也会被匹配并设置 CSS 类名。

想要链接使用“精确匹配模式”，则使用 `exact` 属性。在上面的例子中，“Home”路由就必须以精确匹配模式，否则它的 tab 被点中后，Home 的 tab 会始终保持“激活”状态。

```
<!-- 这个链接只会在地址为/的时候被激活 -->
<router-link :to="{name:'Home'}" exact>
```

3.5 History 的控制

另外有一点需要附加说明，当我们在使用 HTML5 的 History 模式的时候，每次路由的改变都会被“推”（push）到导航历史中保留，在某些情况下我们并不需要浏览器这样做，而是希望它能将原有的记录进行替换，那么我们就需要了解<router-link>是如何通过编程方式控制路由进行导航的。首先 Vue 实例内有一个\$router 对象，这个对象会提供三个方法，<router-link>则是用两种属性来对应这三个方法的调用：

router 的方法	属 性	说 明
push()	—	默认调用此方法
append()	append	将目标 URL 追加到当前 URL 下
replace()	replace	以目标 URL 替换现有的 URL

设置 replace 属性的话，当点击时，会调用 router.replace()而不是 router.push()，于是导航后不会留下 History 记录。

```
<router-link :to="{ name: 'Home' }" replace></router-link>
```

设置 append 属性后，则在当前（相对）路径前添加基路径。例如，我们从/a 导航到一个相对路径 b，如果没有配置 append，则路径为/b，如果配置了，则为/a/b。

```
<router-link :to="{ path: 'relative/path' }" append></router-link>
```

你可能会对此感到一些疑惑，到底这个 URL 的替换与追加有什么实际的作用与意义呢？举一个非常简单的例子你就可以理解了，如果你的 Vue 程序运行于微信客户端，按照用户一般的使用习惯，如果要回退到上一页，就会点击微信左上角的“返回”按钮，这个时候 History API 就会起作用，它就会默认返回至上一次执行 push 的那条历史路径上。如果导航至一个修改个人信息的页面，然后这个页面下又有其他子页，在这些子页中如果执行了一次 push 返回到个人信息页，那么用户点击左上角的“返回”按钮就不是向上一个页返回，而是返回到子页内，显然这并不是我们想看到的。简单点说，push、append 和 replace 是直接控制访问路由在 History 上历史记录的插入和更新方式的，如果用户点击浏览器的前进与后退，就会激发浏览器从这个 History 中查找下一个路由的位置是什么。

最后，在命令行内键入：

```
$ npm run dev
```

就可以看到我们所需要的运行效果了。

如果你足够好奇，对 Tab 上的任意图标点击右键，然后点击“查看”的话，你会惊奇地

发现图片的地址与我们写在代码中的完全是不一样的。以 `home.svg` 为例，在浏览器中可能会是这样一个地址 `/dist/home.svg?65bc5719926d723e279698eaa58d7f49`。其实，你不需要感到惊讶，这要归功于 `webpack` 给我们做的编译处理。在 `<template>` 中显式引用的图片会被 `webpack` 的加载规则所识别，这个规则在 `webpack.config.js` 中可以找到：

```
module: {
  loaders: [
    // ... 省略
    {
      test: /\. (png|jpg|gif|svg) $/,
      loader: 'file',
      query: {
        name: '[name].[ext]?[hash]'
      }
    }
  ]
}
```

这个意义在于，我们不需要再去在意程序引入了哪些资源，在发布时应该对这些资源进行哪些处理，因为 `webpack` 已经为我们做了。

3.6 关于 Fallback

由于我们将路由配置成 `History` 模式，假如用户点击 `Home` 上的 `<router-link>` 时，浏览器的地址栏就会自动改变成对应的 URL (`http://localhost/home`)。如果我们直接在浏览器输入 `http://localhost/home`，你会惊奇地发现浏览器会出现 `404` 的错误！

这是由于直接在浏览器输入 `http://localhost/home`，浏览器就会直接将这个地址请求发送至服务器，先由服务器处理路由，而客户端路由的启动条件是要访问 `/index.html`，这样的话客户端路由就完全失效了！

解决的办法是将所有发到服务端的请求利用服务端的 `URLRewrite` 模板重新转发给 `/index.html`，启动 `VueRouter` 进行处理，而浏览器地址栏的 URL 保持不变。

这个问题在开发期是不会出现的，因为我们在开发环境中使用的是 `webpack` 的 `DevServer`，`DevServer` 是对这个问题进行了处理的，只要打开 `webpack.config.js`，找到 `devServer` 配置属性就可以见到：

```
// ...
```



```
devServer: {  
  historyApiFallback: true  
},
```

而当我们部署到生产环境时,就需要在 Web 服务器上进行一些简单配置以支持 Fallback 了。

Apache

如果使用 Apache 就要在它的配置文件内加入以下 URLRewrite 模块的配置:

```
<IfModule mod_rewrite.c>  
  RewriteEngine On  
  RewriteBase /  
  RewriteRule ^index\.html$ - [L]  
  RewriteCond %{REQUEST_FILENAME} !-f  
  RewriteCond %{REQUEST_FILENAME} !-d  
  RewriteRule . /index.html [L]  
</IfModule>
```

Nginx

Nginx 则更加简单,当出现 404 时将自动重定向至 index.html:

```
location / {  
  try_files $uri $uri/ /index.html;  
}
```

Node.js (Express)

如果使用 Node.js 作为服务端的话,可以安装一个 Fallback 插件以支持此功能,可以到 <https://github.com/bripekens/connect-history-api-fallback> 下载并安装此插件。

其他后端程序

如果你使用的是 Python 或者 Ruby on Rails 这一类后端程序,单纯修改 Web 服务端的设置是不够的,因为 Nginx 或者 Apache 会将请求通过语言解释插件转发至 Python 或者 Rails 的处理程序,由它们的路由系统去判定应如何操作,所以我们只能在后端程序中加入一些特殊的处理以支持 Fallback。

Flask(Python)

如果使用 Flask 的话,增加 Fallback 会比较简单,只要增加一个全局的错误捕获装饰器进行重定义即可:

```
from flask import Flask, render_template
```

```
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.app_errorhandler(404)
def api_fall_back(e):
    return index()
```

Ruby on Rails

以下是 Rails 的 Fallback 支持，假定 index 页面在 HomeController 下的是 Action，那么在路由文件内的设置将是这样的：

```
# ~/config/routes.rb
root 'home#index'

# ....

match 'path*', :to 'home#index'
```

注意

一旦我们进行了上述的配置，你的服务器就不再返回 404 错误页面，因为对于所有路径都会返回 index.html 文件。为了避免发生这种情况，应该在 Vue 应用里面覆盖所有的路由情况，然后再给出一个 404 页面。

```
const router = new VueRouter({
  mode: 'history',
  routes: [
    { path: '*', component: NotFoundComponent }
  ]
})
```

或者，如果用 Node.js 开发后台，可以使用服务端的路由来匹配 URL，当没有匹配到路由的时候返回 404，从而实现 Fallback。

3.7 小结

本章的内容虽短，但却融合了我多年 Web 项目开发的经验，无论项目的大与小，路由

定义必然是最重要的第一件工作。思维导图中画出的站点地图是一种纯粹的逻辑思维过程，也可以说是一个最简单的设计过程。这个过程设计的是 Web 程序的边界和用户导航的路径，我们必须确保有足够的页面来完成相应的工作流程，每个页面之间都能顺畅地互相导航，否则就会出现死链或者死节点的情况。而路由定义则可以切实地实现这一蓝图，从一开始就将要开发的页都做出来，内容可以是空的，但在将它运行到浏览器中时应该确保每个页面之间都能与我们设计的网站地图的导航方式一致，因为这是确定业务流程与导航流程是否一致的一种最佳实践。

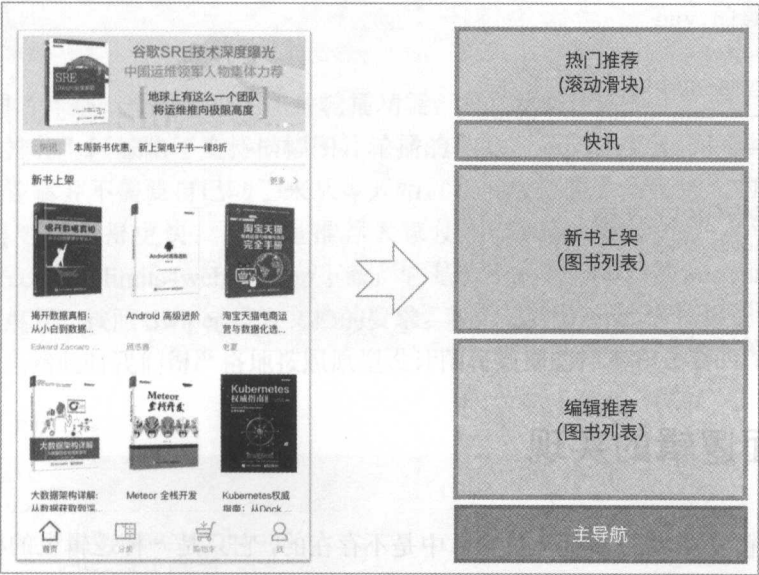
第 4 章 页面的区块化与组件的封装

组件式开发是 Vue.js 的开发基础，同时也是我们在工程化开发时对功能的抽象与重用的根本。所谓的组件化就是将复杂的、充满重复交互界面的组件逐步细化与抽象为简单的、单一化的一个过程。

区块的划分

我们做前端开发都是从上而下地进行设计与布局，如果按功能或者内容分类来对整个页面进行划分的话，你会很自然地将一个页面的内容分为一个或多个功能区，事实上这是人们的阅读习惯。从这种习惯入手，我们可以很容易将一个复杂的页面划分为功能单一的 Vue 组件，划分区块的目的就在于将复杂问题简单化，将一个抽象的设计工作分解为具体的开发工作。

本章将通过组件化的思维来设计与构建 Home 页，也会着重于实践，通过实践领悟个中的理论比纯粹讲如何使用 Vue 会有意思得多。我们先从设计图入手，将 Home 的页面结构从功能区块上来进行划分：



这是一个从具象化到抽象化的基本过程，如果没有这一过程，我们根本没有办法将这些功能与实际工作相结合，继而进行工作的细分与实现步骤的安排。

首页上的“新书上架”和“编辑推荐”明显是由两个功能相同的组件构成的，那么我们可以将其视为两个类型相同的功能组件，然后就可以得到以下几个构成 HOME 页的组件。

功 能 区	组 件	命 名
热门推荐	滚动滑块	slider
快讯	快讯	announcement
新书上架, 编辑推荐	图书列表	book-list

主导航除了在四个顶层页面内显示，不会在其他页面或组件内使用，在实现上只需要写在 Main.vue 内即可，可见它是没有什么重用性需求的，我们并不需要对其进行组件化。

接下来我们分别将 slider、announcement 和 book-list 写成 Vue 的组件并用来装配 Home 页面。

我们一定要养成从框架入手的良好编程习惯，写代码就像是在画画一样，一开始就应该从打草稿开始，然后慢慢地给草稿添加各种细节，多次细化后最终才完成这部作品。所以我们先不用关心这些组件如何来写，先将文件按照之前的命名约定先创建出来，组件的模板和上一章的页面模板一样。

```
└─ src
   │  └─ App.vue
   │  └─ Main.vue
   │  └─ assets
   │  └─ components
   │     │  └─ Announcement.vue
   │     │  └─ Booklist.vue
   │     └─ Slider.vue
   └─ Category.vue
   └─ Home.vue
   └─ Me.vue
   └─ Shoppingcart.vue
   └─ main.js
```

4.1 页面逻辑的实现

正如我们前文所说，“页面”在 Vue 中是不存在的，它只是一种逻辑上的概念。事实上，

“页面”这个概念就是由 DOM 元素和一个甚至多个自定义 Vue 组件复合而成的复合型组件。我们应该如何分清楚哪些部分使用 DOM 元素实现,哪些部分又应该封装为更小级别的 Vue 组件呢?

首先,我们要从原型设计图入手,可以先从功能性布局上划分出对应的功能区域,如前面的设计图所示。

然后用 HTML 的注释标记作为页面上的“区域占位”,先给页面搭一个最基本的结构,这就像作画时先给整体打草稿勾一个轮廓一样。当我们一步一步地将这些细节描绘出来后,再将这些“轮廓线”从画中抹掉。

```
<template>
  <div>
    <div class="section">
      <div>
        <!-- 热门推荐 -->
        <!-- 快讯 -->
      </div>
    </div>
    <div class="section">
      <!-- 新书上架 -->
    </div>
    <div class="section">
      <!-- 编辑推荐 -->
    </div>
  </div>
</template>
```

“热门推荐”是一个最常见的图片轮播功能,而且这是一款基于面向手机的应用,所以这个“热门推荐”区域除了支持横幅图片轮播的功能,还应该支持手势滑动换页的功能。这样的实现逻辑并不需要自己动手来从零开始,我们应该学会站在别人的肩膀上做开发,这样才走得更远走得更快。在这里推荐大家使用 Swiper 这个组件,这个组件可以在 <https://github.com/nolimits4web/swiper> 下载,它是一个具有 9000 多个 star 的代码库!可见其受欢迎程度了。按照 Swiper 官方文档的要求,我们先将 Swiper 所需要的 HTML 格式和样式编写好,当然此时我们得严格地按照原型设计图将数据的样本和必要的图片资源准备好。

代码如下所示。

```
<template>
  <div>
    <div class="section">
      <!-- 热门推荐 -->
```



```

<div class="swiper-container"
  ref="slider">
  <div class="swiper-wrapper">
    <div class="swiper-slide">
      
    </div>
  </div>
  <div class="swiper-wrapper">
    <div class="swiper-slide">
      
    </div>
  </div>
  <div class="swiper-pagination"
    ref="pagination">
  </div>
</div>
<!-- 快讯 -->
</div>
</div>
<div class="section">
  <!-- 新书上架 -->
</div>
<div class="section">
  <!-- 编辑推荐 -->
</div>
</div>
</template>

```

接下就要在代码中引入对 `swiper-container` DOM 元素应用 `Swiper` 这个对象了。在它的官方文档中，`Swiper` 的使用是通过 CSS 选择器声明将 `Swiper` 应用到那个页面元素上的：

```
const swiper = new Swiper('.slider-container')
```

如果我们直接将它抄过来，应用到我们的组件代码中会出现问题。一个好的组件应该是与外部没有依赖关系的，或者说依赖关系越少越好，这叫低耦合。如果我们用 CSS 选择器作为 `Swiper` 定位页面上元素依据的话，假如在一个页面上同时有两个 `slider-container`，那么这个组件就会乱套！

所以，我们应该避免用这种模糊的指定方式，而应该使用 `Vue.js` 提供的更精确的指明方式在元素中添加 `ref` 属性，然后在代码内通过 `this.$refs` 引用名来引用。

这是 `Vue.js 2.0` 后的变化，`ref` 标记是标准的 HTML 属性，它取代了 `Vue.js 1.x` 中 `v-ref` 的写法。如果你曾是 `Vue.js 1.x` 的开发者，那么必须要留意这一点，`v-ref` 已经被废弃了！

```

<script>
import Swiper from "swiper"           // 引入 Swiper 库
import 'swiper/dist/css/swiper.css' // 引入 Swiper 所需要的样式

export default {
  // 不要选用 created 钩子而应该采用 mounted
  // 否则 Swiper 不能生效, 因为 created 调用时元素还没挂载到 DOM 上
  mounted () {
    new Swiper(this.$refs.slider, {
      pagination: this.$refs.pagination,
      paginationClickable: true,
      spaceBetween: 30,
      centeredSlides: true,
      autoplay: 2500,
      autoplayDisableOnInteraction: false
    })
  }
}
</script>

```

组件化的过程就是在不断地对代码进行去重与抽象封装的过程, 上述代码中<div class="swiper-container"></div>元素内所包含的内容除了的 src 属性内的数据是不同的, 其他的都是重复的, 很明显这些图片的地址应该是从服务器中传过来的。首先我们将这些重复的图片地址先放到 data 属性内并重新改写上述代码。其次, 考虑到用户点击当前显示的轮播图片时应该跳转到图书的详细页面内, 那么这个 slides 内存储的就不单单是一个图片地址, 应该还要有一个图书 ID, 用于作为路由跳转的参数:

```

export default {
  data () {
    return {
      slides:[
        { id:1, img_url:'./fixtures/sliders/t2.svg'},
        { id:2, img_url:'./fixtures/sliders/t2.svg'}
      ]
    }
  },
  // ...省略
}

```

用 v-for 指令标签对 slides 数组列表进行渲染:

```

<div class="swiper-wrapper"
  v-for="slide in slides">

```

```

<router-link class="swiper-slide"
  tag="div"
  :to="{ name: 'BookDetail', params:{ id: slide.id} }">
  
</router-link>
</div>

```

那么，“热门推荐”区域的功能就实现完成了，代码从一堆被“压缩”成一小段了！slides 中的数据我们先暂时写死，后面我们再将数据进行统一的处理。

“快讯”区域的实现比较简单，思路与实现同“热门推荐”是一样的，先按原型图直接编写 HTML，然后将应该服务器提取的数据部分抽取到 data 中，最后重构页面。

```

<div class="announcement">
  <label>快讯</label>
  <span>{{ announcement }}</span>
</div>

```

data 的定义代码：

```

export default {
  data () {
    return {
      announcement: '今日上架的图书全部 8 折',
      slides: [
        { id: 1, img_url: './fixtures/sliders/t2.svg' },
        { id: 2, img_url: './fixtures/sliders/t2.svg' }
      ]
    }
  },
  // ... 省略
}

```

4.2 封装可重用组件

接下来是“新书上架”和“编辑推荐”这两个主要的图书列表了。我们还是用前文的办法，先按照原型图来依葫芦画瓢，准备好样本数据和图书封面实现代码：

```

<div class="book-list">
  <div class="header">
    <div class="heading">最新更新</div>
    <div class="more">更多...</div>

```

```

</div>
<div class="book-items">
  <div class="book">
    <div class="cover"></div>
    <div class="title">揭开数据真相：从小白到数据分析达人</div>
    <div class="authors">Edward Zaccaro, Daniel Zaccaro</div>
  </div>
  <div class="book">
    <div class="cover"></div>
    <div class="title">Android 高级进阶</div>
    <div class="authors">顾浩鑫</div>
  </div>
  <div class="book">
    <div class="cover"></div>
    <div class="title">淘宝天猫电商运营与数据化选品完全手册</div>
    <div class="authors">老夏</div>
  </div>
  <div class="book">
    <div class="cover"></div>
    <div class="title">大数据架构详解：从数据获取到深度学习</div>
    <div class="authors">朱洁, 罗华霖</div>
  </div>
  <div class="book">
    <div class="cover"></div>
    <div class="title">Meteor 全栈开发</div>
    <div class="authors">杜亦舒</div>
  </div>
  <div class="book">
    <div class="cover"></div>
    <div class="title">Kubernetes 权威指南：从 Docker 到 Kubernetes 实践
全接触（第2版）</div>
    <div class="authors">龚正, 吴治辉, 王伟, 崔秀龙, 闫健勇</div>
  </div>
</div>
</div>
<div class="book-list">
  <div class="header">
    <div class="heading">编辑推荐</div>
    <div class="more">更多...</div>
  </div>
  <div class="book-items">
    <div class="book">
      <div class="cover"></div>
      <div class="title">自己动手做大数据系统</div>
    </div>
  </div>

```



```

        <div class="authors">张粤磊</div>
      </div>
      <div class="book">
        <div class="cover"></div>
        <div class="title">智能硬件安全</div>
        <div class="authors">刘健皓</div>
      </div>
      <div class="book">
        <div class="cover"></div>
        <div class="title">实战数据库营销——大数据时代轻松赚钱之道（第 2 版）</div>
      </div>
      <div class="book">
        <div class="cover"></div>
        <div class="title">大数据思维——从掷骰子到纸牌屋</div>
        <div class="authors">马继华</div>
      </div>
      <div class="book">
        <div class="cover"></div>
        <div class="title">从零开始学大数据营销</div>
        <div class="authors">韩布伟</div>
      </div>
      <div class="book">
        <div class="cover"></div>
        <div class="title">数据化营销</div>
        <div class="authors">龚正, 吴治辉, 王伟, 崔秀龙, 闫健勇</div>
      </div>
    </div>
  </div>

```

这样的代码是不是很难看？大量的重复逻辑存在于代码内。这并不要紧，正如上文提到的，这是一个勾勒轮廓的过程，重复性的内容就可以被提取出来封装成一个或多个组件，但封装之前我们得知道向这个组件输入一些什么样的数据，这个组件应该具有什么样的行为或者事件。我们先从提取数据入手，将上面的内容提取成两个数组对象，然后将多个重复性的元素用列表循环取代：

```

{
  latestUpdated:[
    {
      "id": 1,
      "title": "揭开数据真相：从小白到数据分析达人",
      "authors": [ "Edward Zaccaro", "Daniel Zaccaro" ],
      "img_url": "1.svg"
    }
  ]
}

```



```
},
{
  "id": 2,
  "title": "Android 高级进阶",
  "authors": [
    "顾浩鑫"
  ],
  "img_url": "2.svg"
},
{
  "id": 3,
  "title": "淘宝天猫电商运营与数据化选品完全手册",
  "authors": [
    "老夏"
  ],
  "img_url": "3.svg"
},
{
  "id": 4,
  "title": "大数据架构详解：从数据获取到深度学习",
  "authors": [
    "朱洁",
    "罗华霖"
  ],
  "img_url": "4.svg"
},
{
  "id": 5,
  "title": "Meteor 全栈开发",
  "authors": [
    "杜亦舒"
  ],
  "img_url": "5.svg"
},
{
  "id": 6,
  "title": "Kubernetes 权威指南：从 Docker 到 Kubernetes 实践全接触（第2版）",
  "authors": [
    "龚正",
    "吴治辉",
    "王伟",
    "崔秀龙",
    "闫健勇"
  ],
}
```

```

    "img_url": "6.svg"
  }
  ],
  recommended:[ ... ] //内容结构与 latestUpdated 相同，在此略过
]
}

```

用 v-for 标签渲染上述的数据：

```

<div class="section">
  <div class="book-list">
    <div class="header">
      <div class="heading">新书上架</div>
      <div class="more">更多...</div>
    </div>
    <div class="book-items">
      <div class="book"
        v-for="book in latestUpdated">
        <div class="cover">
          
        </div>
        <div class="title">{{ book.title }}</div>
        <div class="authors">{{ book.authors | join }}</div>
      </div>
    </div>
  </div>
</div>
<div class="section">
  <div class="book-list">
    <div class="header">
      <div class="heading">编辑推荐</div>
      <div class="more">更多...</div>
    </div>
    <div class="book-items">
      <div class="book"
        v-for="book in recommended">
        <div class="cover">
          
        </div>
        <div class="title">{{ book.title }}</div>
        <div class="authors">{{ book.authors | join }}</div>
      </div>
    </div>
  </div>
</div>

```

经过第一次的抽象，代码减少了很多，但是这两个 Section 内显示的内容除了标题与图书的数据源不同，其他的逻辑还是完全相同的。也就是说，它们应该是由一个组件渲染的结果，只是输入参数存在差异，那么就还存在一次融合抽象的可能。此时我们就可以动手将这两个列表封装成为一个 BookList 组件。

先对原页面的内容进行重构，预先命名 BookList 组件，接着确定在页面上的用法。这很重要，Home 页面就是 BookList 的调用方，BookList 的输入属性是与 Home 之间的接口，当接口被确定了，组件的使用方式也同样被固定下来了。

```
<div class="section">
  <book-list :books="latestUpdated"
    heading="最新更新">
</book-list>
</div>
<div class="section">
  <book-list :books="recommended"
    heading="编辑推荐">
</book-list>
</div>
```

标记名称被确定，类名与文件名也就被确定了，先在 Home 页中引入 BookList 组件：

```
// 按照工程结构约定，组件放置在 components 目录
import BookList from "../components/BookList.vue"

export default {
  data () {
    announcement: '今日上架的图书全部 8 折',
    slides: [
      { id: 1, img_url: './fixtures/sliders/t2.svg' },
      { id: 2, img_url: './fixtures/sliders/t2.svg' }
    ],
    latestUpdated: [ ... ], // 这两个数组内容太多，为了便于阅读此处略去具体定义
    recommended : [ ... ]
  },
  components: {
    BookList
  },
  ...
}
```

这里通过 import 导入组件定义，用 components 注册自定义组件，注意对引入的组件名

称要采用**大驼峰命名法**。在 Vue.js 的官方文档中是这样约定的：“所有引入的组件在 `<template>` 内使用时都以小写形式出现，如果类名由两个大写开头的单词所组成，那么在第二个大写字母前面需要添加“-”来与之前的单词进行分隔。”我们按照这个使用约定来构建 `<template>` 内的视图内容。

组件与标记的对应关系如下表所示。

组件注册名称	模块标记
BookList	<code><book-list></code>

以上这点必须谨记，否则 Vue 将不能识别注册的自定义组件。

接口与用法都确定了我们就能开始真正地编写 BookList 组件了，在 `components` 目录下创建一个 `BookList.vue` 的组件文件：

```
export default {
  props: [
    'heading', // 标题
    'books'    // 图书对象数组
  ],
  filters: {
    join(args) {
      return args.join(',')
    }
  }
}
```

要向组件输入数据就不能使用 `data` 来作为数据的容器了，因为 `data` 是一个内部对象，此时就要换成 `props`。

我们可以通过“作用域”来理解 `data` 和 `props`，`data` 的作用域是仅仅适用于内部而对于外部的调用方是不可见的，换句话说它是一个私有的组件成员变量；而 `props` 是内部外部都可见，是一个公共的组件成员变量。

将之前提取的 HTML 内容放置其中，并用 `props` 定义的属性替换原有的数据对象。另外，这里定义了一个 `join` 过滤器，用于将 `authors`（作者）数组输出为以逗号分隔的字符串，改写后的组件模板如下：

```
<template>
  <div class="book-list">
    <div class="header">
      <div class="heading">{{ heading }}</div>
      <div class="more">更多...</div>
```



```
</div>
<div class="book-items">
  <div class="book">
    v-for="book in books">
      <div class="cover">
        
      </div>
      <div class="title">{{ book.title }}</div>
      <div class="authors">{{ book.authors | join }}</div>
    </div>
  </div>
</div>
</template>
```

4.3 自定义事件

BookList 组件的封装可以说是基本成形了，按照设计图，当用户点击某一本图书之时要弹出一个预览的对话框，如下图所示。



也就是说，每个图书元素要响应用户的点击事件，显示另一个窗口或对话框显然应该由 Home 页进行处理，所以需要 BookList 在接收用户点击事件后，向 Home 组件发出一个事件通知，然后由 Home 组件接收并处理显示被点击图书的详情预览。

在第 1 章我们就介绍过如何通过 v-bind 指令标记接收并处理 DOM 元素的标准事件。此时需要更深入一步，就是为 BookList 定义一个事件，并由它的父组件，也就是 Home 页接收并进行处理。

Vue 的组件发出自定义事件非常简单，只要使用 \$emit("事件名称") 方法，然后输入事件名称就可以触发指定“事件名称”的组件事件，具体如下所示。

```
<div class="book"
  v-for="book in books"
  @click="$emit('onBookSelect', book)">
  <div class="cover">
    
  </div>
  <div class="title">{{ book.title }}</div>
  <div class="authors">{{ book.authors | join }}</div>
</div>
```

\$emit 的第一个参数是事件名称，第二个参数是向事件处理者传递当前被点击的图书的具体数据对象。完成这一步后，BookList 组件的封装工作就宣告结束，可以回到 Home 页中加入由 BookList 组件所发出的 onBookSelect 事件了。

\$emit() 是 Vue 实例的方法，在 <template> 内所有调用的上下文都将默认指向 Vue 组件本身(this)，所以无须声明，但如果是在代码内调用的话则需要通过 this.\$emit 方式显式引用。

在 Home 中增加一个 preview(book) 的方法用来显示图书详情预览的对话框，preview 方法可以先不实现，我们会将其留在下文中进行处理。

```
export default {
  data () {
    // ... 省略
  },
  methods: {
    preview (book) {
      alert("显示图书详情")
    }
  },
  // ...
}
```

接收自定义事件与接收 DOM 事件的方式是一样的，也是使用 `v-bind` 指令标记接收 `onBookSelect` 事件：

```
<book-list :books="latestUpdated"
  heading="最新更新"
  @onBookSelect="preview($event)">
</book-list>
```

为什么这里会出现一个 `$event` 参数呢？其实这个参数是被 Vue 注入到 `this` 对象中的，当事件产生时，这个 `$event` 参数用于接收由 `$emit('onBookSelect', book)` 的第二个传出参数 `book`。每个采用 `v-bind` 指令接收的事件都会自动产生 `$event` 对象，如果事件本身没有传出的参数，那么这个 `$event` 就是一个 DOM 事件对象实例。

4.4 数据接口的分析与提取

至此，我们已完成了 Home 页中布局所需要的基本元素与组件了。接下来就是要重构 `data` 内的数据了，现在的数据是被写死在 `data` 内的。我们需要将这些数据变活，让它们从服务端获取。

我们先来回顾一下完整的 `data` 的结构，`[...]`在此表示略去，实际代码应该写成数组：

```
data () {
  return {
    announcement: '今日上架的图书全部 8 折',
    slides: [ ... ],
    latestUpdated: [ ... ],
    recommended : [ ... ]
  }
}
```

如果要接入到服务端，在 Home 初始化时就应该自动从服务端获取 `announcement`、`slides[]`、`latestUpdated[]`和 `recommended[]`四个参数，这里我们已经在不知不觉中设计出了 HOME 页的前端与服务端交互数据接口，现在的 `data` 内容正是这个数据接口。我们先将这些数据抽出来放到一个 `~/fixtures/home/home.json` 文件内，然后将 `data` 内的数据清空：

```
data () {
  return {
    announcement: '',
    slides: [],
    latestUpdated: [],
```

```

      recommended : []
    }
  }
}

```

最后，将与服务端通信的数据接口和 API 的用法保存到~/fixtures/home/README.md，
以下是 API 文档的样本：

请求地址

```
HTTP GET '/api/home'
```

返回对象

```

{
  announcement: ''           // 快讯的内容
  slides:[                   // 热门推荐图书
    {
      id: 1,                  // 图书编号
      img_url:'/assets/banners/1.jpg' // 滑块图地址 大小
    }
    //...
  ],
  latestUpdated: [           // 新书上架
    {
      id:1,                   // 图书编号
      title:'BookName',       // 书名
      img_url:'/assets/covers/1.jpg', // 封面图地址
      authors:["作者 1", ... , "作者 n"], // 作者列表
    }
    // ...
  ],
  recommended : []           // 编辑推荐，对象定义与 latestUpdated 相同
}

```

在多人协作开发的情况下采用 Vue 架构的前端开发都很自然地会与后端开发独立，或者说是齐头并进式地并行式开发更为贴切。要确保前后端开发的一致性最关键的是控制接口。因为它们是前端与后端关键结合点，API 接口的任何变化都会导致前端与后端代码的修改甚至是进行新的迭代。

由于前端是消化用户需求的第一站，所以由前端来制定接口是最合适不过的了。因此，我在团队协作式开发过程中最重要的关键任务就是制作上述的这一份 API 接口说明文件，只有它被确立之后才能真正地实现前后端的协作式开发。

对于较小的项目我们的做法是将所有的文档保存到项目根目录下的 docs 内，同时也纳入到 Git 的源码管理中，方便平时查阅。而对于规模较大的项目我们会使用 GitBook 编写一份更加完整的手册，文档在设计时编写是最容易的，如果到项目验收时才补充一定会有疏漏，不要让文档成为开发人员的技术债务。

4.5 从服务端获取数据

有了文档的定义，接下来就要实现从/api/home 这个地址上获取数据了。Vue 的标准库并没有提供访问远程服务器（AJAX）功能，所以我们需要安装另一个库——vue-resource。vue-resource 并不是 Vue 官方提供的库，而是由 Pagekit (<https://github.com/pagekit>) 团队所开发的，它的体积小，学习成本低，是 Vue 项目中用于访问远程服务器的一个优秀的第三方库。

在讲述 vue-resource 之前，先用最传统的方法来获取数据，然后再用 vue-resource 改写这个过程。这样做的目的是不想打断我们现在的编程思路，因为 vue-resource 的使用还涉及它的安装与配置等用法，因此先用 jQuery.ajax 的方式来编写这个数据获取的方法。

在 Home 页的 created 钩子内加入以下代码来获取 data 内的数据：

```
export default {
  data () {
    return {
      announcement:'',
      slides:[],
      latestUpdated: [],
      recommended : []
    }
  },
  created () {
    var self = this
    $.get('/api/home').then(res => {
      self.announcement = res.announcement
      self.slides = res.slides
      self.latestUpdated = res.latestUpdated
      self.recommended = res.recommended
    })
  }
  // ... 省略
}
```

如果使用 jQuery 的话就需要引入 jQuery 内很多我们并不需要的内容（我们根本就不直接操作 DOM，jQuery 在此一点用处都没有）。这样将增大编译后的文件大小，最终发布包

的大小会影响下载速度，从而降低用户的使用体验。其次，从上述代码中可见，我们需要用一个 `self` 变量来“hold”住当前的 `Vue` 对象实例，这未免让代码显得很糟糕。而用 `vue-resource` 这个库的话，就可以规避掉使用 `jQuery` 所带来的这两个坏处。

- `vue-resource` 插件具有以下特点：
- 体积小——`vue-resource` 非常小巧，压缩以后大约只有 12KB，服务端启用 `gzip` 压缩后只有 4.5KB 大小，这远比 `jQuery` 的体积要小得多。

支持主流的浏览器——和 `Vue.js` 一样，`vue-resource` 除了不支持 IE9 以下的浏览器，其他主流的浏览器都支持。

支持 `Promise API` 和 `URI Templates`——`Promise` 是 ES6 的特性，`Promise` 的中文含义为“承诺”，`Promise` 对象用于异步计算。`URI Templates` 表示 `URI` 模板，有些类似于 `ASP.NET MVC` 的路由模板。

支持拦截器——拦截器是全局的，拦截器可以在请求发送前和发送请求后做一些处理。拦截器在一些场景下会非常有用，比如请求发送前在 `headers` 中设置 `access_token`，或者在请求失败时，提供共通的处理方式。

安装

我们可以用以下命令将 `v-resource` 安装到本地的开发环境中：

```
$ npm i vue-resource -D
```

`vue-resource` 是一个 `Vue` 的插件，在安装完成后需要在 `main.js` 文件内载入这个插件，代码如下所示。

```
import Vue from 'vue'
import VueResource from 'vue-resource'

Vue.use(VueResource)
```

对于那些不能处理 `REST/HTTP` 请求方法的老旧浏览器（例如 IE6），`vue-resource` 可以打开 `emulateHTTP` 开关，以取得兼容的支持：

```
Vue.http.options.emulateHTTP = true
```

通常 `RESTful API` 的一个约定俗成的规则是 `API` 的地址都以 `/api` 或 `/rest` 为资源根目录，我们在此也采用此约定。为了在调用时省下更多的代码，我们可以在 `Vue` 的实例配置内对 `HTTP` 进行配置：

```
new Vue({
  http: {
    root: '/api', // 指定资源根目录
```



```
headers: {} // 添加自定义的 http 头变量
},
// ... 省略
})
```

`headers` 参数用于对发出的请求的头内容进行重写与自定义，例如加入验证信息或者代理信息等。

使用 `use` 方法引入 `vue-resource` 后，`vue-resource` 就会向 `Vue` 的根实例“注入”一个 `$http` 的对象，那么我们就可以在所有 `Vue` 实例内通过 `this.$http` 来引用它，它的用法与 `jQuery` 几乎一样，很容易上手。将前文的代码使用 `vue-resource` 来改写：

```
export default {
  data () {
    return {
      announcement: '',
      slides: [],
      latestUpdated: [],
      recommended : []
    }
  },
  created () {
    // HTTP GET /api/home
    this.$http.get('/home').then(res=> {
      this.announcement = res.body.announcement
      this.slides = res.body.slides
      this.latestUpdated = res.body.latestUpdated
      this.recommended = res.body.recommended
    })
  },
  ...
}
```

`vue-resource` 的一个最大的好处是它会自动为我们在异步成功调用返回后将 `Vue` 实例注入到回调方法中，这样我们就不需要额外地去用另一个变量来“hold住”`this`了。

我们还可以让代码变得更加简洁一些：

```
this.$http.get('/api/home')
  .then((res) => {
    for prop in res.body {
      this[prop] = res.body[prop]
    }
  }, (error) => {
    console.log(`获取数据失败: ${error}`)
  })
```

附加说明: \$http API 参考

对应常用的 HTTP 方法, vue-resource 在 \$http 对象上提供了以下包装方法:

- get(url, [options])
- head(url, [options])
- delete(url, [options])
- jsonp(url, [options])
- post(url, [body], [options])
- put(url, [body], [options])
- patch(url, [body], [options])

options 对象参考:

参 数	类 型	描 述
url	String	请求的 URL
body	Object, FormData, string	写入请求 body 属性的数据对象 (一般用于发送表单对象)
headers	Object	重写发出请求的 HTTP 头对象变量
params	Object	用作生成带参数 URL 的参数对象
method	String	HTTP 方法 (例如 GET, POST, ...)
timeout	Number	单位为毫秒的请求超时时间 (0 表示无超时时间)
before	function(request)	请求发送前的处理函数, 类似于 jQuery 的 beforeSend 函数
progress	function(event)	上传数据时的 ProgressEvent 事件的处理函数 (用于计算上传进度)
credentials	Boolean	是否应使用凭据进行跨站点访问控制请求
emulateHTTP	Boolean	发送 PUT、PATCH、DELETE 请求时以 HTTP POST 的方式发送, 并设置请求头的 X-HTTP-Method-Override
emulateJSON	Boolean	将 request.body 的内容以 application/x-www-form-urlencoded 编码方式发送

回调参数 response 对象参考:

属性说明

属 性	类 型	描 述
url	String	原请求的 URL 地址
body	Object, Blob, string	响应对象的 body 内容
headers	Header	响应的请求头对象
ok	Boolean	响应的 HTTP 状态码在 200~299 之间时, 该属性为 true
status	Number	响应的 HTTP 状态码
statusText	String	响应的状态文本

方法说明

方 法	类 型	描 述
text()	Promise	以 string 形式返回 response.body
json()	Promise	以 JSON 对象形式返回 response.body
blob()	Promise	以二进制形式返回 response.body，多用于从响应对象中直接以流的方式读取文件内容或图片数据

4.6 创建复合型的模板组件

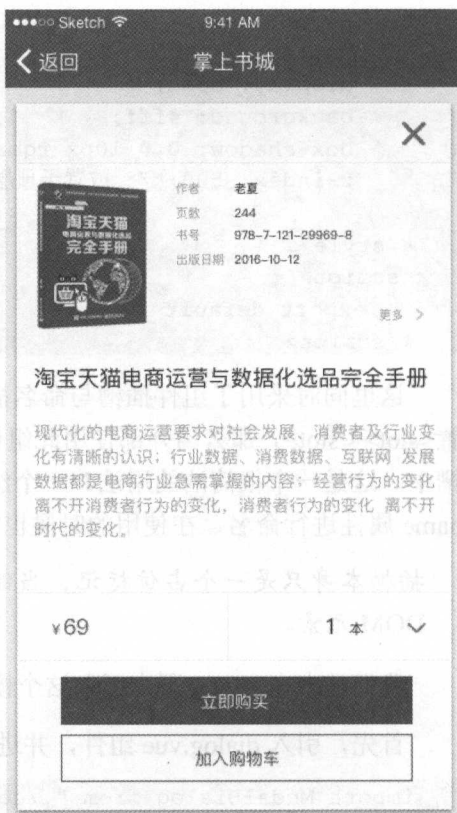
Home 页组件内还有一个方法没有实现，那就是 `preview()`，也就是当用户点击图书时弹出的一个模态窗口，如右图所示。

对于这个应用场景，应用之前先创建一个组件页，然后加入代码再重构的实现思路显然不可行，我们必须先实现一个模态窗口组件才能在其上放置显示图书详情的元素以及实现添加购物车和立即购买的功能。

这个模态窗口组件有一个特殊的地方，就是它自身是一个容器，我们需要在这个容器所提供的特定区域内放置其他的 DOM 元素或者组件。此时我们可以认为模态窗口组件就是一个复合型组件。Vue 可以通过一特殊的指令标记来在组件模板内“划”出一个独立的区域让调用方（父组件）可以向模态窗口组件中插入新的代码，以扩充其功能。这种对外提供插入能力的指令就是所谓的“插槽”，也就是 `<slot>`。

新建一个 `src/components/dialog.vue` 文件，具体内容如下：

```
<template>
  <div class="dialog">
    <div>
      <!-- 头部及标题 -->
```



```

        <slot name="header"></slot>
      </div>
    <div>
      <!-- 内容区域 -->
      <slot></slot>
    </div>
  </div>
</template>
<style>
  .dialog {
    position: absolute;
    top: 24px;
    left: 24px;
    right: 24px;
    bottom: 24px;
    display: none;
    background: #fff;
    box-shadow: 0 0 10px rgba(0,0,0,.5);
    z-index: 500; /* 放置于顶层 */
  }
</style>
<script>
  export default {}
</script>

```

这里同时采用了组件插槽与命名插槽两种方式，默认插槽也就是直接在组件模板内放置`<slot></slot>`，那么当外部使用此组件时，在组件标记内的元素都会被自动插入到默认插槽中，这是一个很简洁的用法！一个组件只能拥有一个默认插槽，其他的插槽则需要采用`name`属性进行命名，在使用的时候也需要对插槽进行声明。

插槽本身只是一个占位标记，当组件渲染时，`<slot></slot>`标记自身并不会输出任何 DOM 元素。

我们马上在 Home 页内引入这个模态窗口组件来试试它的用法。

首先，引入 `dialog.vue` 组件，并进行子组件注册操作：

```

import ModalDialog from "../components/dialog.vue"
export default {
  ...
  components: {
    ModalDialog
  }
}

```

然后在<template>内加入<modal-dialog>:

```
<template>
  <div>
    ...
    <modal-dialog>
      <div slot="header">此处是 header 插槽的内容</div>
      <div>这个 DIV 将自动默认插槽的内容</div>
    </modal-dialog>
  </div>
</template>
```

我们将 modal-dialog 设计为默认情况下是不显示的, 所以我们需要给它加入一些方法, 让 Home 页能通过编程方式对其进行显示或隐藏的控制。

在 dialog.vue 组件中加入 open 和 close 方法对:

```
export default {
  data () {
    return {
      is_open : false
    }
  },
  methods: {
    open() {
      if (!this.is_open) {
        // 触发模态窗口打开事件
        this.$emit('dialogOpen')
      }
      this.is_open = true
    },
    close() {
      if (this.is_open) {
        // 触发模态窗口关闭事件
        this.$emit('dialogClose')
      }
      this.is_open = false
    }
  }
}
```

在 CSS 中加入.open 样式类:

```
<style>
.dialog { ... }
```



```
.dialog.open {
  display: block;
}
</style>
```

最后在 dialog.vue 的顶层元素内加入 class 属性开关切换：

```
<template>
  <div class="dialog-wrapper"
    @class="{ 'open': is_open }">
  </div>
</template>
```

模态对话框组件就宣告完成了。

以下为 dialog.vue 完整代码：

```
<template>
  <div class="dialog-wrapper"
    :class="{ 'open': is_open }">
    <div class="overlay" @click="close"></div>
    <div class="dialog">
      <div class="heading">
        <slot name="heading"></slot>
      </div>
      <slot></slot>
    </div>
  </div>
</template>
<script>
import "../dialog.less"
export default {
  data () {
    return {
      is_open: false
    }
  },
  methods: {
    open () {
      if (!this.is_open) {
        // 触发模态窗口打开事件
        this.$emit('dialogOpen')
      }
      this.is_open = true
    },
```

```

        close() {
            if (this.is_open) {
                // 触发模态窗口关闭事件
                this.$emit('dialogClose')
            }
            this.is_open = false
        }
    }
}
</script>

```

样式表 dialog.less 的代码:

```

.dialog-wrapper {
    &.open {
        display: block;
    }
    height: 100%;
    display: none;
    &>.overlay {
        background: rgba(0, 0, 0, 0.3);
        z-index: 1;
        position: absolute;
        left: 0px;
        top: 0;
        right: 0;
        bottom: 0;
    }

    &>.dialog {
        z-index: 10;
        background: #fff;
        position: fixed;
        top: 24px;
        left: 24px;
        right: 24px;
        bottom: 24px;
        padding: 24px 14px;
        box-shadow: 0 0 10px rgba(0, 0, 0, .8);
        & heading {
            padding: 12px;
        }
    }
}

```

接下来在 Home 页组件对 modal-dialog 内增加引用声明和事件处理:

```
<modal-dialog ref="dialog"
  @dialogClose="selected=undefined">
  <div slot="header">
    <div class="dismiss"
      @click.prevent="$refs.dialog.close()"></div>
  </div>
  <div>
    
  </div>
  <div>
    {{ selected.title }}
  </div>
  ...
</modal-dialog>
```

最后完成 preview 方法:

```
export default {
  data () {
    return {
      // ... 省略
      selected:undefined
    },
  },
  methods: {
    preview (book) {
      this.selected = book
      this.$refs.dialog.open()
    },
    // ... 省略
  },
  // ... 省略
}
```

4.7 数据模拟

虽然 Home 组件的代码实现已经完成,而且已经通过 vue-resource 接入了与服务端通信的功能,但现在是一个纯前端的开发环境,并没有可以被访问的后端服务,那么如何让我们的程序获取服务端的数据呢?此时我们就需要运用另一种技术来解决这个问题了,这个技术就是“数据模拟”(或者称数据仿真)。

“数据模拟”就是用一个对象直接模拟服务端的实现返回的数据结果，而这些数据结果是我们预先采样收集来的，与真实运行数据几乎是一样的。通过数据模拟保证前端程序即使在没有服务端支持的情况下也能运行。

在前文中已做了一个小小的铺垫，就是将 `data` 的样本数据抽取出来保存到了 `~/fixtures/home/home.json` 文件中。为了能先让开发环境运行起来，我们可以加入一些助手类来模拟实际的运行数据。

我们需要定义一个获取模拟数据的对象 `faker`，在项目中所有模拟数据都通过它来获取。

```
// ~/fixtures/faker.js
import HomePageData from "../home.json"

var slider_images = require.context('./sliders', false, /\. (png|jpg|gif|svg)$/ )
var cover_images = require.context('./covers', false, /\. (png|jpg|gif|svg)$/ )

HomePageData.top.forEach((x) => {
  x.img_url = slider_images('./' + x.img_url)
})

HomePageData.promotions.forEach((x) => {
  x.img_url = cover_images('./' + x.img_url)
})

export default {
  getHomeData() {
    return HomePageData
  }
}
```

这个 `faker` 使用了一个动态加载图片的技巧，将一个指定目录下的所有文件全部加载到一个模块方法中，然后通过具体名称返回它在 `webpack` 编译加载后的真实地址。不使用 `../assets/sliders/图片.png` 相对路径的方式引用，是因为 `webpack` 将程序编译并加载到开发服务器后，这些图片地址的真实路径并不是指向 `~/src/assets` 目录的，因此我们要用 `require.context` 函数将编译后的资源作为一个模块加载进来，然后再通过名称获取其正确的地址。

```
var slider_images = require.context('./sliders', false, /\. (png|jpg|gif|svg)$/ )

slider_images('./1.png') // => 获取真正的 /sliders/1.png 的地址
```


在 home 页面组件中，在 created 钩子方法内加入 faker，我们可以加入一个开关变量 debug 用于判断当前运行环境是否为开发环境，如果是则使用数据模拟方式获取数据。

```
import faker from "../fixtures/faker"

// 判断当前环境是否是开发环境
const debug = process.env.NODE_ENV !== 'production'

export default {
  data () {
    // ... 省略
  },
  created () {
    if (debug) {

      const fakeData = faker.getHomeData()
      for prop in fakeData {
        this[prop] = fakeData[prop]
      }

    } else {
      this.$http.get('/api/home')
        .then((res) => {
          for prop in res.body {
            this[prop] = res.body[prop]
          }
        }, (error) => {
          console.log(`获取数据失败: ${error}`)
        })
    }
  },
  // ... 省略
}
```

现在我们就可以在终端运行 `$ npm run dev` 查看本示例的完整运行效果了。

4.8 小结

将本章中整个示例的实现过程画成一个工作流程图的话，你将会很清楚开发一个页面

组件应该执行哪些步骤了：



(1) **依葫芦画瓢**——拿到界面设计图后无须思考太多，先用框架圈出功能区块，然后直接编写视图的 HTML。

(2) **代码去重**——将视图模板中不断重复的逻辑封装为组件，减少页面的重复逻辑。

(3) **抽取数据结构**——将页面中的文字用数据对象与数组取代，并制定数据结构的说明文档。

(4) **采集与制作样本数据**——参照数据结构说明文档采集更多的真实样本，切忌胡乱地敲入一些字符，在数据不明确的情况下可能会遮盖一些本应很明显的用户需求。

(5) **分析设计组件接口**——简化组件的使用接口，让组件变得更好用。

(6) **组件内部的细化与重构**——优化组件的内部实现，使其变得更合理。

4.9 扩展阅读：Vue 组件的继承——mixin

Vue 开发是一种面向组件的开发，而面向组件开发的本质即是面向对象。既然是面向对象就离不开抽象、封装与继承三大基本特性。其实在前文中多处提及的众多的组件编写方法与分析，总结起来也不过是不断地对组件进行抽象与封装，力求使每个组件能尽量与外部保持一定的独立性，以达到自容纳（Self contains）和服务自治（Self services）的效果，这样做的目的就是最大限度地增加组件的可重用性，减少代码的重复。

三大特性中的继承却一点没有提及，JavaScript 一直被很多初学者诟病面向对象能力差，事实并非如此！自从原型模式被完全引入 ES 后，JS 就具有完整的面向对象能力，并由于弱类型语言的特性令其开发的灵活度更优于一些传统的纯面向对象语言。在 ES6 的语言特性改善后就更为强大，ES6 已经可以和一些面向对象语言一样编写类和进行类之间的继承。虽然 Vue2 官方推荐我们采用 ES6 作为开发的主语言，但实质上并没有在 Vue 中应用语言级别的继承用法，而是选择了另一种聪明的做法：混合。

混合（mixins）

混合是一种灵活的分布式复用 Vue 组件的方式。混合对象可以包含任意组件选项。以

组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项。之所以说“混合”是一种聪明的做法，是因为这种方式更适合于 JS 的开发思维。首先，继承虽然是面向对象的三大基本特性之一，也是极为常用的构造类库的方法，可惜的是它往往被滥用。例如，当继承深度超过三代以后，类族就会变得极为庞大，子类中往往存在大量毫无作用的祖先类中遗留的特性或者方法，可以想象到这样的类库必然臃肿不堪难以维护。其次，JS 天生就是个弱类型语言，强类型化的继承方式给 JS 的开发带来的麻烦会比较多。如果既要使用继承又希望避开由于继承造成的复杂的多态性，复合/混合是一种非常不错的解决方案，这个概念在 Ruby 中也很常用。所以说我非常喜欢 Vue 采用混合的方式来实现公共特性的共用而不是采用继承。

在我参与的几个面向商业应用的 Vue 项目中，使用到“混合”的场景其实并不多，这是因为通过复合型的 Vue 组件已经可以去除掉大量的重复性，而在开发一些基础性的界面的套件时，“混合”就显得很重要了。例如，在 v-uikit 项目 (<http://www.github.com/dotnetage.com/vue-ui>) 中就遇到这样的场景，当开发列表控件 uk-list 和下拉列表控件 uk-dropdown-list 时，发现它们的界面实现完全不同，但代码逻辑却是相同的。首先，来看看 uk-list 控件原来的代码：

```
<template>
  <ul :class="{
    'uk-list':true,
    'uk-list-line':showLine,
    'uk-list-striped':striped,
    'uk-list-space':space
  }">
    <li v-for="item in listItems"
      @click.prevent="selectItem(item)">{{ item[textField] }}</li>
  </ul>
</template>
<script>
export default {
  props: {
    showLine: {
      type: Boolean,
      default: false
    },
    space: {
      type: Boolean,
      default: false
    },
    striped: {
```

```
    type: Boolean,
    default: false
  },
  items: {
    type: Array,
    default: () => []
  },
  textField: {
    type: String,
    default: 'label'
  },
  valueField: {
    type: String,
    default: 'value'
  },
  data () {
    return {
      selectedItem: undefined
    }
  },
  computed: {
    selectedValue () {
      return this.selectedItem ? this.selectedItem[this.valueField] : ''
    },
    listItems () {
      if (this.items && this.items.length) {
        const t = typeof(this.items[0])
        if (t === 'string' || t === 'number') {
          return this.items.map((i) => {
            const obj = {}
            obj[this.textField] = i
            obj[this.valueField] = i
            return obj
          })
        }
      }
    }
  },
  return this.items
},
methods: {
  selectItem(item) {
    this.selectedItem = item
  }
}
```

```

      this.$emit('selectedChange', item)
    }
  }
}
</script>

```

这个控件是将一个 JS 数组用 UIkit 的列表样式展现出来，支持点击选择并发出 `selectedChange` 事件，以提供给其他的界面控件使用。

然后是 `uk-dropdown-list`，这个控件将一组下拉列表包装至任意的容器类元素上，使其具有一个下拉菜单的功能。例如在下拉件组内放入一个按钮，点击这个按钮时在其下方就会出现一个下拉菜单。这个组件的代码如下：

```

<template>
  <div data-uk-dropdown="{mode:'click'}"
    class="uk-button-dropdown">
    <slot></slot>
    <div class="uk-dropdown">
      <ul class="uk-nav uk-nav-dropdown">
        <li v-for="item in listItems"
          :class="{ 'uk-nav-header': item.isHeader}">
          <a
            @click.prevent="selectItem(item)">{{ item[textField] }}</a></li>
        </ul>
      </div>
    </div>
</template>
<script>
export default {
  props: {
    items: {
      type: Array,
      default: () => []
    },
    textField: {
      type: String,
      default: 'label'
    },
    valueField: {
      type: String,
      default: 'value'
    }
  },
  data () {

```



```

    return {
      selectedItem: undefined
    }
  },
  computed: {
    selectedValue () {
      return this.selectedItem ? this.selectedItem[this.valueField] : ''
    },
    listItems () {
      if (this.items && this.items.length) {
        const t = typeof(this.items[0])
        if (t === 'string' || t === 'number') {
          return this.items.map((i) => {
            const obj = {}
            obj[this.textField] = i
            obj[this.valueField] = i
            return obj
          })
        }
      }

      return this.items
    }
  },
  methods: {
    selectItem(item) {
      this.selectedItem = item
      this.$emit('selectedChange', item)
    }
  }
}
</script>

```

这两个组件在交互处理的逻辑上有很大一部分是相同的，或者说它们的控制部分应该是从一个组件中继承下来的。这个时候我们就可以用 Vue 的 mixins 实现这种功能性的混合。首先将两个控件中完全相同的部分提取出来，做成一个 BaseListMixin.js 的组件：

```

export default {
  props: {
    items: {
      type: Array,
      default: () => []
    },
    textField: {

```



```

    type: String,
    default: 'label'
  },
  valueField: {
    type: String,
    default: 'value'
  }
},
data () {
  return {
    selectedItem: undefined
  }
},
computed: {
  selectedValue () {
    return this.selectedItem ? this.selectedItem[this.valueField] : ''
  },
  listItems () {
    if (this.items && this.items.length) {
      const t = typeof(this.items[0])
      if (t === 'string' || t === 'number') {
        return this.items.map((i) => {
          const obj = {}
          obj[this.textField] = i
          obj[this.valueField] = i
          return obj
        })
      }
    }

    return this.items
  }
},
methods: {
  selectItem(item) {
    this.selectedItem = item
    this.$emit('selectedChange', item)
  }
}
}

```

然后将 `uk-list` 和 `uk-dropdown-list` 中相同的代码删除, 用 `mixins` 引入 `BaseMixinList` 类, 这样在 `BaseMixinList` 中定义的属性 (`props`)、方法 (`methods`)、计算属性等所有的 `Vue` 组件内允许定义的字段都会被混合到新的组件中, 其效果就如类继承。

uk-list 的代码就变为:

```
<script>
import BaseListMixin from './BaseListMixin'
export default {
  mixins: [BaseListMixin],
  props: {
    showLine: {
      type: Boolean,
      default: false
    },
    space: {
      type: Boolean,
      default: false
    },
    striped: {
      type: Boolean,
      default: false
    }
  }
}
</script>
```

ul-dropdown-list 的代码变为:

```
<script>
import BaseListMixin from './BaseListMixin'
export default {
  name: 'UkDropdown',
  mixins: [BaseListMixin]
}
</script>
```

混合比继承好的地方就是一个 Vue 组件类可以与多个不同的组件进行混合 (mixins 是一个数组, 可以同时声明多个混合类), 复合出新的组件类。而大多数的继承都是单根模式 (从一个父类继承) 的, 同时由于 JS 是弱类型语言, 语言解释引擎并不需要强制地了解每个实例来源于哪一个类才能进行实例化, 由此就产生了无限的可能和极大的组件构型的灵活性。

第 5 章 Vue 的测试与调试技术

我从事软件开发 10 多年来一直倡导敏捷开发，从测试驱动开发到行为驱动式开发，我始终都是一名践行者。很多开发者都认为测试对于项目来说是可有可无的，甚至不将测试工作纳入到开发进度中。

其实，测试是一个非常美妙的世界，一旦进入根本停不下来！

对于 Vue 这一类重度采用前端技术的项目而言，测试更加是一个开发的加速器。代码写完了我们要确认是否符合要求，始终得一边运行一边测试或者调试。一个功能点可能需要调试十几次甚至上百次，每次都做相同的人工操作，再有耐心的人都会随便填写一些测试数据，在逻辑上跑通就算过关了。这也导致软件发布后，在生产环境中出现了各种“不可预知”的问题。而最坏的情况是，如果手工重现这些问题并且修复后，遇到项目迭代要重新检测程序是否能正常运作，你还能记得这些问题的操作过程和输入的数据吗？

我经常看到不少的前端开发人员基本上都只进行人工测试，屏幕左边开一个编码窗口，右边打开一个浏览器，左边写代码右边看效果。当然这种编码场景我也用，但仅限于制作或者调试界面样式和页面布局的时候，而不是从开始到最终都这样做！支持“热加载”是让我喜欢上 Vue 开发的其中一个原因，因为有了这个功能，无论是编写样式又或者需要用视觉检查功能时都将无比高效。我更愿意让程序帮我检查代码是否正确，虽然程序是自己写的，但人是会出错的，何况只是用视觉判断程序的正确性，出错几乎是无可避免的。其次，我们总会遇到这样一种情况，一个项目完成上线后就去忙其他项目了，突然某天收到一个紧急的命令要在原来那个程序上加点功能或者修改一个小 bug，如果没有测试，谁遇到这种情况都心慌，甚至连改程序的勇气都没有，谁说得准程序一改又会出现什么错误呢？而且还得凭着记忆去将以前做过的人工测试重新做一次才能安心。

测试不单单是一个质检员，它还是一个异常备忘录，甚至可以说是一个为我们保驾护航的好助手。

你曾试遇到过修改代码后，导致其他地方出现问题的时候吗？相信绝大多数程序员都遇到过。因为这几乎是不可避免的，特别是在规模庞大的代码面前，代码与代码之间可能是环环相扣的，改变一处会影响另一处。

但如果这种情况不会发生呢？如果你有一种方法能知道改变后会出现的结果呢？这无疑极好的。因为修改代码后无须担心会破坏什么东西，从而程序出现 bug 的概率更低，在 debug 上花费的时间更少。

这就是单元测试的魅力，它能自动检测代码中的任何问题。在修改代码后进行相应的测试，若有问题，能立刻知道问题是什么，问题在哪和正确的做法是什么。这可以完全消除任何猜测！

5.1 Mocha 入门

在开始介绍 Vue 的单元测试之前，我们需要做一些基本知识的准备，第 2 章介绍了 vue-cli 脚手架建立的单元测试骨架的组成与运行原理。仅仅了解这些内容是不够的，编写单元测试之前必须了解测试框架的用法和与之配套的编程工具的使用方法。

基本的测试骨架

单元测试文件都放在 test/unit/specs 目录下，这是在第 2 章就已经定下的工程目录使用约定，且每个测试文件要以 *spec.js 文件名结尾。创建 test/unit/specs/array.spec.js 文件，写一个对 JavaScript 的 Array 对象的基本功能的测试示例，只有实践才是快速学习的捷径。

每个测试案例文件都会遵循以下基本模式。首先，有个 describe 块：

```
describe('Array', () => {  
  // 测试序列  
})
```

describe 用于把单独的测试聚合在一起，在 TDD 中称之为测试序列 (Suite)，也可以将它看作功能分组。第一个参数用于指示测试什么，第二个参数是一个匿名函数。这里我们先来举一个最简单的例子，在本例中，由于我们打算测试 Array 功能，那么将以 Array 这个对象的名称作为测试序列的名称。

序列的嵌套

测试序列是一种分组方式，它也允许以树状结构对子序列进行嵌套，从而可以将一个大的测试序列划分为更小的组成部分。例如：

```
describe('User', () => {  
  describe('Address', () => {
```

```
// ...  
})  
})
```

测试用例

测试序列内至少有一个 `it` 块，否则 Mocha 会忽略测试序列内的内容，不会执行任何的动作。例如：

```
describe('Array', () => {  
  it('应该在初始化后长度为 0', () => {  
    // 这里编写测试代码的实现  
  })  
})
```

`it` 用于创建实际的测试，它在 TDD 中被称为测试用例 (Test-Case)。其第一个参数是对该测试的描述，且该描述的语言应该是我们日常用语的句式（而非编程语言）。测试用例用 `it` 作为函数名，其用意就是希望通程序引导开发人员用书面语言去描述测试用例的作用，如：“It should be done with ...”，或者 “It should be have some value” 等。直接翻译成我们中国人的句式就将变成：“应该…输出 XXX 结果”，或者 “应该…完成 XXX 操作” 这样的句式。

这是一种偏向于行为式的描述方式（虽然 Mocha 号称是支持行为式驱动测试的框架，然而其只能属于类行为式测试，而非真正的行为驱动式测试框架，关于行为式驱动开发在下文自有交待，在此暂且放下），对于单元测试来说可以将其归类为一种“一般性的通用描述”。顾名思义，单元测试的对象是某个特定的类或者模板，因此 `describe` 内才直接用类名进行描述，那么 `it` 内最应该用的描述方式是“方法名”或“属性名”。仍然以 `Array` 为例（因为它的方法属性 JS 程序员应该都懂）：

```
describe('Array', () => {  
  it('#slice', () => {  
    // ...  
  })  
  
  it('#join', () => {  
    // ...  
  })  
})
```

为什么用方法名或者属性名是最好的呢？因为好的程序应该是能达到自描述的，如果从名称上都看不出其用法，那么是否就可以从使用上验证这个方法或属性的命名有问题而需要重构呢？这不正是写测试的其中一种目的所在吗？

所有 Mocha 测试都以同样的骨架编写，虽然它还有其他的写法，但上述写法是一种推荐用法，所以我们都应该遵循这个相同的基本模式：

- (1) 测试序列用类名命名。
- (2) 当测试用例用于测试指定方法或属性默认效果时用“#+成员名”方式命名。
- (3) 当测试用例的测试内容不能归属于某个方法或属成员时用“应该…输出 XXX 或应该…完成 XXX 操作”的句式陈述。

在某些情况下我们希望 Mocha 跳过功能测试，那么我们可以使用 `xdescribe` 函数取代 `describe`，这样 Mocha 将不会执行它们，而单纯将其视为“跳过”的状态，同理 `it` 也可以用 `xit` 来表达忽略执行。

这是写好单元测试的一个重点，“思路决定行为”，文字性的内容表达准确清楚才能得到正确的测试结果。文字性表述就是测试的架构设计，这也是为何要耗费这么多文字来论述这个文字性描述规则的缘由。

断言

现在我们已经知道如何组织与编写测试用例了，接下来就需要对测试的结果进行判断，我们称这一过程为“断言”(Assertion)。Mocha 自身并没有配置断言库，在第 2 章中我们也了解到，`vue-cli` 的 `webpack` 模板已经通过 `Karma` 为我们的测试框架配置了 `Sinon-Chai` 这个库了，它基于 `Sinon` 和 `Chai` 两个库的合成优化版本，所以我们并不需要引入其他的断言库了。当前的测试用例的上下文 `this` 内被注入了 `Chai`，所以可以直接使用 `Chai` 提供的标准断言语法。其他的断言库还有很多，也可以按自己的喜好定制，在此就不做过多的表述，毕竟以一书之篇幅也难以尽述。

`Chai` 断言库有两种语法，一种是 `expect` 语法，另一种是 `should` 语法，两种语法只是在表达顺序上略有不同，`expect` 语法则更加通用，毕竟这些都是仿照“`Rspec`”做出来的，我们就采用正统的最佳实践。

具体的句式是 `expect([实际被检测值]).to` 语法，其表达的语法都是期待一个“实际”(向 `expect` 传递的参数)值等于一个“期待”值。

上例中测试 `Array` 的初始值应为空，即我们需要创建一个数组并确保它为空：

```
describe('Array', () => {
  it('应该初始化为空数组', () => {
    var arr = []
    expect(arr).to.be.lengthOf(0)
  })
})
```

```
  })  
  })
```

实际值是测试代码的结果，期待值是预想的结果。由于数组的初始值应为空，因此，在该案例中的期待值是 0。

上述的 `to` 是一个陈述式的链式接口，它们只是为了让断言更加容易理解，将它们添加进断言中使句子变得啰唆但是增加了易读性，它们并不会提供任何测试功能：

- `to`;
- `be`;
- `been`;
- `is`;
- `that`;
- `and`;
- `have`;
- `with`;
- `at`;
- `of`;
- `same`;
- `a`;
- `an`。

紧跟在上述这些链式接口后的才是断言方法，Chai 中的断言方法非常多，下文中还补充了 `Sinon-Chai` 为仿真测试而加入的其他的一些基于 `Sinon` 的断言。为了保持阅读的一致性，Chai 的断言 API 放在了本书的“附录 A——Chai 断言参考”内，当你开始使用断言时它们就在那里等着你。

钩子

在功能测试内，每个测试场景运行在一个独立的进程内，测试之间是不应该存在依赖关系的。但是经常会出现这样的情况，多个场景之间可能会执行同样的初始化操作，或者测试完成后的变量或数据清理工作。面对这些情况，`Mocha` 提供了 4 个钩子函数在 `describe` 内进行统一的调用。

- `beforeEach`——在每个场景测试执行之前执行；
- `afterEach`——在每个场景执行完成之后执行；
- `before`——在所有场景执行之前执行（仅执行一次）；
- `after`——在所有场景执行之后执行（仅执行一次）。

```

describe("Array", () => {
  var expectTarget = []

  beforeEach(() => {
    expectTarget.push(1)
  });

  afterEach(() => {
    expectTarget = []
  });

  it("应该存有一个为 1 的整数", () => {
    expect(expectTarget[0]).toEqual(1)
  });

  it("可以有多个的期望值检测", () => {
    expect(expectTarget[0]).toEqual(1)
    expect(true).toEqual(true)
  });
});

```

例如，我们可以创建一个 Vue 实例，在各个测试用例中共享：

```

describe("UkButton", () => {
  let vm = undefined

  beforeEach(() => {
    vm = new UkButton({propsData:{
      color:'primary'
    }}).$mount()
  })

  afterEach(() => {
    vm.destroy()
  })

  it("设置 Button 的颜色", () => {
    expect(vm.$el.getAttribute('class')).toEqual('uk-button
uk-button-primary')
    vm.componentOptions.propsData.color = 'success'
  })

  it("Button 的颜色应该被改成了 success", () => {
    expect(vm.$el.getAttribute('class')).toEqual('uk-button
uk-button-success')
  })
}

```

```

    })
  })

```

异步测试

Vue 代码中会在很多情况下出现异步调用，例如上传、API 调用、加载一个外部资源等，对这类代码进行测试时就需要异步测试用例的支持。Mocha 的异步测试用法与 Jasmine 一模一样，就是在 `it` 内加入一个 `done` 函数，在所有的断言执行完成后调用 `done()` 就可以释放测试用例并告知 Mocha 测试的执行结果。

例如，我们有一个 `User` 对象，这个对象具有一个 `save` 方法，这个方法通过 `AJAX` 将数据保存到服务端。如果服务端没有返回错误，那么我们就认为这个 `save` 方法是成功的，具体的代码如下所示。

```

describe('User', () => {
  describe('#save()', () => {
    it('应该成功保存到服务端且不会返回任何错误信息', done => {
      const user = new User('Luna')
      user.save( err => {
        if (err) done(err) // 如果返回错误码直接将错误码输出至控制台
        else done()
      })
    })
  })
})

```

将上述代码写得更简洁一点，可以将 `done` 作为回调参数使用：

```

describe('User', () => {
  describe('#save()', () => {
    it('应该成功保存到服务端且不会返回任何错误信息', done => {
      const user = new User('Luna')
      user.save(done)
    })
  })
})

```

使用 Promises 的另一种替代方案就是将 Chai 断言作为 `it` 的返回值，将 Chai 断言作为一个 Promises 对象返回让 Mocha 进行链式处理，但要实现这样的效果，你需要一个叫 `chai-as-promised` 的库支持 (<https://www.npmjs.com/package/chai-as-promised>)。在 Mocha v3.0.0 以后的版本中，如果直接构造 ES6 上的 Promise 对象则会被 Mocha 认为是非法的：

```

it('应该完成此测试', (done) => {
  return new Promise(resolve => {

```

```

    assert.ok(true);
    resolve()
  })
  .then(done)
})

```

这样做的结果将得到如下的异常信息: "Resolution method is overspecified. Specify a callback *or* return a Promise; not both.. In versions older than v3.0.0, the call to done() is effectively ignored."

待定的测试

“待定测试”是实际开发过程中用得很多的一种方式, 测试驱动的开发简单来说就是: 编写失败的测试→实现代码→使测试通过→重构。然而, 在实现过程中你会发现很难分清哪些“失败”的测试是要实现的, 哪些是因为代码有问题无法通过测试而要重构的。为了将它们区分开来, 我们可以将第一步中的“编写失败的测试”调整为“编写待定的测试”, 这样就能很清楚地将它们区分开来。

在 Mocha 中只要我们不向 it 函数传入实现测试的函数 (第二个参数), Mocha 就会默认这个测试是待定的。

```

describe('Array', () => {

  describe('#indexOf()', () => {
    // pending test below
    it('should return -1 when the value is not present');
  })
})

```

在输出时待定的测试显示的字体颜色是黄色的 (失败是红色, 通过是绿色), 这样我们一眼就能看出哪些测试的功能是要去实现的了。

重试

对于端到端测试, 由于要使用 Selenium 向外部或者其他服务发起请求, 而且 Selenium 的运行性能并不高, 很可能导致我们的测试由于超时或者运行过快而出现我们并不希望看到的失败结果。Mocha 提供了一个 `retries` 的方法, 在指定的次数内如果出现失败, Mocha 并不会直接报告测试错误, 而是在指定次数内重新尝试运行测试直至运行成功为止:

```

describe('重试', () => {

  // 指定最大的重试次数为 4 次
  this.retries(4)

```



```

beforeEach(()=>{
  browser.get('http://www.yahoo.com')
})

it('应该在第三次重试后成功', () => {
  this.retries(2)
  expect($('.foo').isDisplayed()).to.eventually.be.true
})
})

```

5.2 组件的单元测试方法

这是一项在 Vue 开发中必须掌握的技能，掌握了组件的单元测试就能独立地运行一个组件，并且测试你编写的所有的方法、属性和事件是否与你的设计相符。而且这些是自动化运行的，运行一个指令就能知道所有被测组件是否正常。如果没有组件单元测试而只是在页面运行时观察组件是否正常，一旦出现错误就很难判断这些错误是由页面引发的还是组件本身所引起的。“保障一台汽车的生产质量得从一颗螺丝钉开始。”

如何对 Vue 组件进行测试

假设我们有以下的一个组件：

```

// ~/src/components/my-component.js
export default {
  template: '<span>{{msg}}</span>',
  props: [ 'msg' ],
  created () => {
    console.log("Created");
  }
}

```

问题

我们应如何测试 my-component 在页面中的实际运行效果？

分析

- (1) 只运行<my-component>组件，在它通过测试前不需要放到真正的页面上运行。
- (2) 只需要检测这个组件最终输出的 HTML 内容就可判定是否通过测试。

```
<my-component msg='你好'></my-component>
```

正确输出的 HTML 应为:

```
<span>你好</span>
```

这就是我们对这个组件的最基本测试需求, 先来建立一个单元测试程序的基本结构:

```
// test/unit/spec/my-component.spec.js
import Vue from 'vue/dist/vue'
import MyComponent from 'components/my-component'

describe('my-component', () => {
  it('$mount()', () => {
    // 此处填写测试代码
  })
})
```

注意: Karma 配置了只加载具有*.spec.js 后缀的文件, 所以我们的测试文件都必须以*.spec.js 结尾, 否则会被 Karma 忽略。

接下来通过 Vue.extend 方法构建一个继承至 VComponent 的测试容器组件, 在 template 属性中直接写<my-component>在 Vue 组件内的真实用法, 然后实例化这个容器组件, 最后从\$el 变量中获取 Vue 最终生成的内容, 具体代码如下:

```
const expectedMsg = '你好'

// 构造测试容器组件
const HtmlContainer = Vue.extend({
  data () {
    return {
      text: expectedMsg
    }
  },
  template: `<my-component :msg="text"></my-component>`,
})

const vm = new HtmlContainer()
expect(vm.$el.querySelector('span').textContent).to.be.eq(expectedMsg)
```

运行这个测试:

```
$ npm run unit
```

输出结果如下图所示。

```

vue-js -- phantomjs - node /usr/local/bin/karma start -- 102x34
RayOSX:vue-js Ray$ karma start
Hash: f4683f5fa2953dc3a97c
Version: webpack 1.13.2
Time: 10ms
webpack: bundle is now VALID.
webpack: bundle is now INVALID.
Hash: 8dc7a7e5f1ae943ddac1
Version: webpack 1.13.2
Time: 306ms
Asset      Size  Chunks             Chunk Names
test/my-component.spec.coffee 198 kB  0 [emitted] test/my-component.spec.coffee
chunk {0} test/my-component.spec.coffee (test/my-component.spec.coffee) 189 kB [rendered]
  [0] ./test/my-component.spec.coffee 502 bytes {0} [built]
  [1] ./bower_components/vue/dist/vue.js 189 kB {0} [built]
  [2] ./app/components/my-component.coffee 75 bytes {0} [built]
webpack: bundle is now VALID.
07 10 2016 00:16:13.283:WARN [karma]: No captured browser, open http://localhost:9876/
07 10 2016 00:16:13.289:WARN [karma]: Port 9876 in use
07 10 2016 00:16:13.290:INFO [karma] Karma v1.3.0 server started at http://localhost:9877/
07 10 2016 00:16:13.290:INFO [launcher]: Launching browser PhantomJS with unlimited concurrency
07 10 2016 00:16:13.295:INFO [launcher]: Starting browser PhantomJS
07 10 2016 00:16:14.560:INFO [PhantomJS 2.1.1 (Mac OS X 0.0.0)]: Connected on socket /#aAHFOt8DmYanMQW
WAAAA with id 52384546

my-component
✓ 应该输出“您好”

PhantomJS 2.1.1 (Mac OS X 0.0.0): Executed 1 of 1 SUCCESS (0.015 secs / 0.011 secs)
TOTAL: 1 SUCCESS

```

在项目代码中我们要追求代码的简洁，其实写测试更需要严格地遵守这一原则，我们应该不惜一切地用更少的代码来完成同样的事情。用这种眼光来看，上述代码就显得有点冗余了。对于这种只读型的组件我们其实可以写得更加简单，根本不需要测试容器，直接构造 MyComponent 实例就够了：

```

const vm = new MyComponent({
  propsData: {
    msg: expectedMsg
  }
})
expect(vm.$el.textContent).to.be.eq(expectedMsg)

```

这样是不是更直接？这里需要注意的是，Vue 组件用 props 定义公共属性，但实例化时传入的却是 propsData，如果你不仔细地阅读 Vue 的官方 API，很可能就会错用这个构造函数了（<https://vuejs.org/v2/api/#propsData>）。

在测试时通过程序直接构造 Vue 实例，引用 Vue 实例时一定要引用 /vue/dist/vue.js 而不能采用 vue.common.js，否则会出现 “You are using the runtime-only build of Vue where the template option is not available. Either pre-compile the templates into render functions, or use the compiler-included build. (found in root instance)” 的警告提示。

创建帮助方法

我们在很多的单元测试中都需要手工创建一个 Vue 的实例入口作为测试容器，这种代码非常冗余，因此我们可以创建一个帮助方法来去除这种重复性。

```
import Vue from 'vue'

export const getVM = (render, components) => {
  return new Vue({
    el: document.createElement('div'),
    render,
    components
  }).$mount()
}
```

有了这个 `getVM` 的帮助方法，我们就可以在单元测试中直接写入渲染的调用逻辑和使用的依赖组件，这样一来就能在很大程度上减轻单元测试的代码量：

```
import {getVM} from '../helper'
import Hello from 'src/Hello.vue'

describe("Render", ()=>{

  it("#mount", ()=> {

    const vm = getVM(<hello>
    </hello>)

    expect(vm.$el.textContent).to.eq('Hello')
  })
})
```

5.3 单元测试中的仿真技术

仿真技术就是用代码工具模拟出实际运行环境中存在的支撑服务，最常用的就是后端服务仿真，在完全没有运行任何后端的情况下“伪造”出一个给测试专用的后端。

那为什么要在测试中运用仿真技术呢？

我们都知道一个完整的 Web 程序必然由后端与前端组成，采用 Vue 这种富前端框架我们可以将前后端的开发独立进行。在这种开发模式下，前后端的开发进度未必是对称的，那就有可能出现前端的某个组件开发完成，它所依赖的后端 API 可能还没有开发出来，此

时前端程序该如何测试呢？还有另一种情况就是后端服务已经存在了并且已投入实际生产运行中，此时的前端开发可能只是一种应用扩展或者升级，一旦需要运行前端进行测试，有可能会向后端发送一些对实际运行毫无用处的数据，甚至会导致数据的混乱。这些情况下测试前端程序就一定要与后端程序脱离，让前端从开发到测试的整个过程完全独立，摆脱所有的外部依赖，此时就需要用仿真技术去模拟这些必要的支持服务了。

为 JS 世界提供仿真技术的最优秀的库就离不开 Sinon，Sinon (<http://sinonjs.org>) 提供了一系列的代码工具帮助你很容易地创建“测试替身”来消除复杂性，像它名字暗示的一样，测试替身用来替换测试中的部分代码。简单来说，Sinon 允许你把代码中难以被测试的部分替换为更容易测试的内容。测试一段代码时，你不希望它被测试以外的部分影响。如果有任何外部因素可以影响测试，那么这个测试就会变得更复杂并且很容易失败。

如果你想测试一段发送 AJAX 的代码，该怎么做呢？你需要运行一个服务器并且确保它返回了测试需要的数据。这种方式导致准备测试环境变得很复杂，同时也给编写和执行单元测试带来很大的不便。如果你的代码依赖于时间（例如重复和超时）又会怎么样呢？假设一段代码在执行操作之前要等待 1 秒钟，该怎么办？你可能会通过 `setTimeout` 来把测试代码的执行延迟 1 秒钟，但这样会导致测试变慢。如果这个等待间隔比 1 秒钟更长呢？比如 5 分钟。我猜你一定不想在每次执行测试代码前都等上 5 分钟。

通过使用 Sinon，我们可以解决以上这些（还有很多其他的）问题，并降低测试的复杂性。

Sinon 有很多功能，但是大部分都是建立在它自身之上的。在掌握了一部分之后，就自然而然地了解下一部分。因此当学习了 Sinon 的基础知识并了解各个组件的功能之后，使用 Sinon 就会变得很容易。

为了让关于这些被调用函数的讨论变得更简单，我会称它们为依赖。我们要测试的方法依赖于另一个方法的返回值。

可以说，使用 Sinon 的基本模式就是使用测试替身替换掉不确定的依赖，例如：

- 当测试 AJAX 时，把 `XMLHttpRequest` 替换为一个模拟发送 AJAX 请求的测试替身。
- 当测试定时器时，把 `setTimeout` 替换为一个伪定时器。
- 当测试数据库访问时，把 `mongodb.findOne` 替换为一个可以立即返回伪数据的测试替身。

由于 JavaScript 是非常灵活的，我们可以把任何方法替换成其他内容。测试替身只不过是把这个想法更进一步罢了。使用 Sinon，我们可以把任何 JavaScript 函数替换成一个测试替身。通过配置，测试替身可以完成各种各样的任务来让测试复杂代码变得简单。

Sinon 将测试替身分分为 3 种类型：

- **Spies**——可以模拟一个函数的实现，检测函数调用的信息。
- **Stubs**——与 Spies 类似，但是会完全替换目标函数。这使得一个被 stubbed 的函数可以执行任何你想要的操作，例如抛出一个异常，返回某个特定值等。
- **Mocks**——通过组合 Spies 和 Stubs，使替换一个完整对象更容易。

此外，Sinon 还提供了其他的辅助方法：

- **Fake timers**——可以用来穿越时间，例如触发一个 `setTimeout`；
- **Fake XMLHttpRequest and server**——用来伪造 AJAX 请求和响应。

有了这些功能，Sinon 就可以解决外部依赖在测试时带来的难题。如果掌握了有效利用 Sinon 的技巧，你就不再需要任何其他工具了。

揭秘 Sinon

Sinon 功能强大，可能看上去很难理解它是如何工作的。为了更好地理解 Sinon 的工作原理，让我们看一些和它工作原理有关的例子。这将有利于我们更好地理解 Sinon 究竟做了哪些工作并在不同的场景中更好地利用它。

我们也可以手工创建 spy、stub 或是 mock。使用 Sinon 的原因在于它使得这个过程更简单了——手工创建通常比较复杂。不过为了理解 Sinon，还是让我们看看如何进行手工创建。

首先，spy 在本质上就是一个函数包装器：

```
// 一个简单的 spy 辅助函数
function createSpy(targetFunc) {
  const spy = () => {
    spy.args = arguments
    spy.returnValue = targetFunc.apply(this, arguments)
    return spy.returnValue
  };

  return spy
}

// 基于一个函数创建 spy
const sum = (a, b) => a + b

const spiedSum = createSpy(sum)

spiedSum(10, 5)
```

```
console.log(spiedSum.args) // 输出: [10, 5]
console.log(spiedSum.returnValue) // 输出: 15
```

使用一个像这样的方法可以很容易地创建 `spy`。但是要明白，`Sinon` 的 `spy` 提供了包括断言在内的丰富得多的功能，这使得使用 `Sinon` 相当容易。

那么 `Stub` 呢？

要创建一个简单的 `stub`，只需把一个函数替换成另一个：

```
const stub = () => { }

const original = thing.otherFunction
thing.otherFunction = stub

// 现在开始，所有对 thing.otherFunction 的调用都会被 stub 的调用所取代
```

但同样需要指出的是，`Sinon` 的 `stub` 有若干优势：

- 包含了 `spy` 的所有功能；
- 可以使用 `stub.restore()` 轻松地恢复原始函数；
- 可以针对 `Sinon stub` 使用断言。

`Mock` 只不过是把 `spy` 和 `stub` 组合在一起，使得可以灵活地使用它们的功能。

虽然 `Sinon` 某些时候看起来使用了很多“魔法”，但大多数情况下，你都可以使用自己的代码实现相同的功能。与自己开发一个库比起来，使用 `Sinon` 只不过是更方便罢了。

5.3.1 调用侦测 (Spies)

`Spies` 是 `Sinon` 中最简单的功能，其他功能都是建立在它之上的。`Spies` 的主要用途是收集函数调用的信息，也可以用它来验证诸如某个函数是否被调用过。

```
const handler = sinon.spy()

// 我们可以像调用函数一样调用一个 spy
handler('Hello', 'World')

// 现在我们可以获取关于这次调用的信息
console.log(handler.firstCall.args);

// 输出: ['Hello', 'World']
```

`sinon.spy()` 返回一个 `spy` 对象。该对象不仅可以像函数一样被调用，还可以收集每次被调用时的信息。在上边的例子中，`firstCall` 属性包含了关于第一次调用的信息，比如 `firstCall.args` 包含了这次调用传递的参数。

虽然可以像上例中一样利用 `sinon.spy` 创建一个匿名 `spy`，但更常见的做法是把一个现有函数替换成一个 `spy`。

```
let user = {
  // ...
  setName (name) {
    this.name = name
  }
}

// 用 setNameSpy 替换掉原有的 setName 方法
const setNameSpy = sinon.spy(user, 'setName')

// 现在开始，每次调用这个方法时，相关信息都会被记录下来
user.setName('Darth Vader')

// 通过 spy 对象可以查看这些记录的信息
console.log(setNameSpy.callCount)
// 输出: 1

// 重要的最后一步，移除 spy
setNameSpy.restore()
```

把一个现有函数替换成一个 `spy` 与前一个例子相比并没有什么特殊之处，除了一个关键步骤：当 `spy` 使用完成后，切记把它恢复成原始函数，就像上边例子中最后一步那样。如果不这样做，你的测试可能会出现不可预知的结果。

在实践中，你可能不会经常用到 `spy`，往往更多地用到 `stub`。但需要验证某个函数是否被调用过时，`spy` 还是很方便的：

```
const myFunction = (condition, callback) => {
  if (condition) {
    callback()
  }
}

describe('myFunction', () => {
  it('should call the callback function', () => {
    let callback = sinon.spy()
```

```
myFunction(true, callback)
assert(callback.calledOnce)
});
});
```

5.3.2 Sinon 的断言扩展

在继续讨论 stubs 之前，让我们快速看一看 Sinon 的断言，使用 spies（和 stubs）的大多数环境中，需要通过某种方式来验证结果。

可以使用任何类型的断言来验证。在上一个关于 callback 的例子中，我们使用了 Chai 提供的 assert 方法来验证值是否为真。

```
assert(callback.calledOnce)
```

这种断言方式的问题在于测试失败时的错误信息不够明确。我们只会得到一条类似“false 不是 true”这样的信息。你可能已经想到了，这样的信息对于确定测试为何会失败并没有什么帮助，我们还是不得不查看测试代码来找到哪里出错了。这可不好玩。

为了解决这个问题，我们可以在断言中加入一条自定义的错误信息。

```
assert(callback.calledOnce, '回调函数尚没有被调用')
```

但是为何不用 Sinon 提供的断言呢？

```
describe('myFunction', () => {
  it('应该调用回调函数', () => {
    const callback = sinon.spy()

    myFunction(true, callback)

    expect(callback).to.have.been.calledOnce()
  });
});
```

像这样使用 Sinon 的断言可以为我们提供一种更加友好的错误信息。当你需要验证更复杂的情况，比如某个函数的调用参数时，这将变得非常有用。

以下是另外一些 Sinon 提供的实用断言：

- `sinon.assert.calledWith` 可以用来验证某个函数被调用时是否传入了特定的参数（这很可能是我最常用的了）；
- `sinon.assert.callOrder` 用来验证函数是否按照一定顺序被调用。

和 spies 一样, Sinon 的断言文档列出了所有可用的选项。如果你习惯使用 Chai, 那么有一个 sinon-chai 插件可供选择, 它可以让你通过 Chai 的 expect 和 should 接口使用 Sinon 的断言。

断 言	说 明
spy.should.have.been.called	应该被调用
spy.should.have.callCount(n)	应该被调用 n 次
spy.should.have.been.calledOnce	应该被调用一次
spy.should.have.been.calledTwice	应该被调用二次
spy.should.have.been.calledThrice	应该被调用三次
spy1.should.have.been.calledBefore(spy2)	应该在 spy2 调用前先被调用
spy1.should.have.been.calledAfter(spy2)	应该在 spy2 调用后被调用
spy.should.have.been.calledWithNew	如果 spy 被 new 运算符调用则返回 true
spy.should.always.have.been.calledWithNew	spy 应该总被构造函数调用
spy.should.have.been.calledOn(context)	spy 至少采用 this 关键字被调用一次
spy.should.always.have.been.calledOn(context)	spy 总是被作为 this 关键字引用调用
spy.should.have.been.calledWith(...args)	函数调用时应该带有指定的参数 args (一个或多个)
spy.should.always.have.been.calledWith(...args)	函数调用时应该总是带有指定的参数 args (一个或多个)
spy.should.have.been.calledWithExactly(...args)	函数调用时应该明确地指定 args 参数 (一个或多个)
spy.should.always.have.been.calledWithExactly(...args)	函数调用时应该总是明确地指定 args 参数(一个或多个)
spy.should.have.been.calledWithMatch(...args)	调用的函数参数必须与指定的 args 参数匹配
spy.should.always.have.been.calledWithMatch(...args)	调用的函数参数必须总是与指定的 args 参数匹配
spy.should.have.returned(returnVal)	应该返回 returnVal 的值
spy.should.always.returned(returnVal)	应该总是返回 returnVal 的值
spy.should.have.thrown(errorObjOrErrorTypeStringOrNothing)	应该抛出指定类型的异常
spy.should.always.thrown(errorObjOrErrorTypeStringOrNothing)	应该总是抛出指定类型异常

如果用 expect 式的语法规则是用 expect(value)取代 spy.should, 例如:

```
const = hello(name, cb) => {
  cb(`hello ${name}`)
}

describe("hello", () => {
  it('应该在回调后输出问候信息', () => {
    const cb = sinon.spy()

    hello('world', cb)

    expect(cb).to.have.been.calledWith('hello world')
```



```
// 或者用 Chai 的 should 语法
cb.should.have.been.calledWith('hello world')
});
});
```

5.3.3 存根 (stub)

由于其灵活和方便, stubs 成为了 Sinon 中最常用的测试替身类型。它拥有 spies 提供的所有功能, 区别在于它会完全替换掉目标函数, 而不只是记录函数的调用信息。换句话说, 当使用 spy 时, 原函数还会继续执行, 但使用 stub 时就不会。

这使得 stubs 非常适用于以下场景:

- 替换掉那些使测试变慢或是难以测试的外部调用;
- 根据函数返回值来触发不同的代码执行路径;
- 测试异常情况, 例如代码抛出了一个异常。

我们可以用类似创建 spies 的方法创建 stubs:

```
const stub = sinon.stub()
stub('hello')
console.log(stub.firstCall.args)

// 输出: ['hello']
```

我们可以创建匿名 stubs, 和使用 spies 时一样, 但只有当你用 stubs 替换一个现有函数时它才开始真正地发挥作用。

举例来说, 如果有一段代码使用了 vue-resource 的 \$http 功能, 那这段代码就会很难被测试。这段代码会向某台服务器发送请求, 你不得不保证测试期间该服务器的可用性。或者你可能会想到在代码里增加一段特殊逻辑以便在测试环境下不会真正地发送请求——这可犯了大忌。在绝大多数情况下你应该保证代码中不会出现针对测试环境的特殊逻辑。

我们可以通过 Sinon 把 \$http 功能替换为一个 stub, 而不是寻求其他糟糕的实现方式。这会使得测试变得很简单。

以下是一个我们要测试的函数。它接受一个对象作为参数, 并通过 \$http 把该对象发送给某个预定的 URL。

```
export default {
```

```

methods: {
  saveUser (user) {
    this.$http.post('/users', {
      first: user.firstname,
      last: user.lastname
    })
  }
}
}

```

通常情况下，由于涉及 AJAX 调用和某个特定的 URL，对它进行测试是比较困难的。但如果我们使用了 **stub**，这就变得很简单。

比方说我们要确保传给 **saveUser** 的回调函数在请求结束后被正确执行。

```

describe('saveUser', () => {
  it('应该在保存成功后进行回调', () => {

    // stub $.post, 这样就不用真正地发送请求
    const post = sinon.stub($, 'post')
    post.yields()

    // 针对回调函数使用一个 spy
    const callback = sinon.spy()

    saveUser({
      firstname: 'Han',
      lastname: 'Solo'
    }, callback)

    post.restore()

    expect(callback).to.have.been.calledOnce
  })
})

```

这里我们把 AJAX 方法替换成了一个 **stub**。这意味着代码里并不会真的发出请求，因此也就不需要相应的服务器了，这样我们就对测试代码里的逻辑取得了完全控制。

由于我们要确保传给 **saveUser** 的回调函数被执行了，我们指示 **stub** 要设置为 **yield**。这意味着 **stub** 会自动执行作为参数传入的第一个函数。这就模拟了 **\$http.post** 的行为——请求一旦完成就执行回调函数。

除了 **stub**，我们还在测试中创建了一个 **spy**。也可以使用一个普通函数作为回调，但是

使用了 spy 后利用 Sinon 提供的 `sinon.assert.calledOnce` 断言可以很容易地验证结果。

在使用 stub 的大多数情况下，可以遵循以下模式：

- (1) 找到导致问题的函数，比如 `$http.post`。
- (2) 观察它是如何工作的以便在测试中模拟它的行为。
- (3) 创建一个 stub。
- (4) 配置 stub 以便按照期望的方式工作。

Stub 不必模拟目标对象的所有行为。只要模拟在测试中用到的行为就够了，其他的都可以忽略。

Stub 的另一个常见使用场景是验证某个函数被调用时是否传入了正确的参数。

例如，针对 AJAX 的功能，我们想验证发送的数据是否正确：

```
describe('saveUser', () => {
  it('应该向指定的 URL 发送正确的参数', () => {

    // 像之前一样为$.post 设置 stub
    const post = sinon.stub(vm.$http, 'post');

    // 创建变量，保存我们期望看到的结果
    const expectedUrl = '/users'
    const expectedParams = {
      first: 'Expected first name',
      last: 'Expected last name'
    }

    // 创建将要作为参数的数据
    const user = {
      firstname: expectedParams.first,
      lastname: expectedParams.last
    }

    saveUser(user, ()=>{})
    post.restore()

    sinon.assert.calledWith(post, expectedUrl, expectedParams)
  })
})
```

同样，我们又为 `$http.post()` 创建了一个 `stub`，但这次我们没有设置它为 `yield`。这是因为此次的测试我们并不关心回调函数，因此设置 `yield` 就没有意义了。

我们创建了一些变量用来保存期望得到的数据——URL 和参数。创建这样的变量是一种不错的做法，因为这样就可以很容易看出这个测试要测哪些数据。我们还可以利用这些值创建 `user` 变量，从而避免重复输入。

这次我们使用了 `sinon.assert.calledWith()` 断言。我们把 `stub` 作为第一个参数传入，因为我们要验证这个 `stub` 被调用时是否传入了正确的参数。

5.3.4 接口仿真 (Mocks)

Mocks 是使用 `stub` 的另一种途径。如果你曾经听过“mock 对象”这种说法，这其实是一码事——Sinon 的 `mock` 可以用来替换整个对象以改变其行为，就像函数 `stub` 一样。

基本上只有需要针对一个对象的多个方法进行 `stub` 时才需要使用 `mock`。如果只需要替换一个方法，使用 `stub` 更简单。

使用 `mock` 时要很小心。由于 `mock` 强大的功能，它很容易导致你的测试过于具体——测试了太多、太细节的内容——这很容易在不经意间导致你的测试变得脆弱。

与 `spy` 和 `stub` 不同的是，`mock` 有内置的断言。你需要预先定义好 `mock` 对象期望的行为并在测试结束前执行验证函数。

比方说我们代码中使用了 `store.js` 来向 `localStorage` 中写入数据，我们希望测试一个与这部分内容相关的函数。我们可以使用一个 `mock` 来协助测试：

```
describe('incrementStoredData', () => {
  it('应该将存储值递增 1', () => {
    const storeMock = sinon.mock(store)
    storeMock.expects('get').withArgs('data').returns(0)
    storeMock.expects('set').once().withArgs('data', 1)

    incrementStoredData()

    storeMock.restore()
    storeMock.verify()
  });
});
```

使用 `mock` 时，我们使用链式调用的方式定义一系列方法以及相应的返回值。除了预

先定义好行为并在测试结束前调用 `storeMock.verify()` 来验证结果，这和使用断言验证测试结果没什么两样。

在 Sinon 的 mock 对象术语中，执行 `mock.expects('something')` 创建了一个预期。例如，函数 `mock.something()` 期望被调用。每一个预期除了 mock 特殊的功能，还支持 spy 和 stub 的功能。

你可能会发现大多数时候使用 stub 比使用 mock 简单得多，这很正常。mock 应该被小心地使用。

最佳实践：使用 `sinon.test()`

无论何时使用 spy、stub 还是 mock，都有一条重要的最佳实践需要牢记。

如果你使用测试替身替换了一个现有函数，记得使用 `sinon.test()`。

在前面的示例中，我们使用了 `stub.restore()` 或 `mock.restore()` 来执行清理操作。这个操作是必要的，否则测试替身会一直存在并给其他测试带来负面影响或是导致错误。

但是直接使用 `restore()` 方法是有问题的。有可能在 `restore()` 执行之前测试代码就因为错误提前结束执行了。

有两种方法可以解决这个问题：把所有的代码放在一个 `try...catch` 块中，这样就可以在 `finally` 块中执行 `restore()` 而不用担心测试代码是否报错。

还有一种更好的方式，就是把测试代码包裹在 `sinon.test()` 中：

```
it('应该用存根做什么', sinon.test(() => {
  const stub = this.stub(vm.$http, 'post')

  doSomething()

  sinon.assert.calledOnce(stub)
}))
```

在上边的示例中，需要注意的是，传递给 `it()` 的第二个参数被包装在 `sinon.test()` 中。另一点要注意的是我们使用的是 `this.stub()` 而不是 `sinon.stub()`。

把测试代码包装在 `sinon.test()` 中后，我们就可以使用 Sinon 的沙盒特性了。它允许我们通过 `this.spy()`、`this.stub()` 和 `this.mock()` 来创建 spy、stub 和 mock。任何使用沙盒特性创建的测试替身都会被自动清理。

注意上边的例子中没有 `stub.restore()` 操作——因为在沙盒特性下的测试里它变得不必要了。

如果在所有地方都使用了 `sinon.test()`，那么就可以避免由于某个测试未能清理它内部的测试替身而导致后续测试随机失败的情况。

5.3.5 后端服务仿真

Sinon 将这种技术称为 **Faker**（骗子），如果直接翻译过来有点贬义，我更喜欢将之称为仿真。

前置仿真也就是请求仿真，这个过程并不会真正地产生 `XMLHttpRequest` 对象，因为这个对象会被 Sinon 产生的 `FakeXMLHttpRequest` 所取代。

```
describe("Home", () => {
  before () {
    this.xhr = sinon.useFakeXMLHttpRequest()
    const requests = this.requests = []
    this.xhr.onCreate = xhr => {
      requests.push(xhr)
    }
  },

  after () {
    this.xhr.restore()
  }

  it("应该从服务器中图书数据", () => {
    const callback = sinon.spy()

    expect(this.requests).toHaveLength(1)

    this.requests[0].respond(200, { "Content-Type": "application/json" },
      '[{"id": 12, "title": "Vue2 实践揭秘"}]')

    expect(callback).to.be.calledWith([{ id: 12, title: "Vue2 实践揭秘" }])
  })
})
```

前置仿真的检测标志在于对请求内容的正确性的检测。

服务端仿真也就是后置仿真，不管前端发出什么样的请求，我们只仿真后端接收请求

的处理。

后置仿真与前置仿真最大的不同之处是它完全让前端产生一个真实的 XMLHttpRequest 对象，在这个对象真正向服务端发出之前进行截获，然后进行结果仿真并返回。

```
describe('Comments', () => {
  before () => {
    this.server = sinon.fakeServer.create()
  },

  after () => {
    this.server.restore()
  },

  it("应该从服务器中获取评论" , () => {
    // 模拟服务器的最终输出效果
    this.server.respondWith("GET", "/some/article/comments.json",
      [200, { "Content-Type": "application/json" },
        '[{"id": 12, "comment": "Hey there" }]' ])

    const callback = sinon.spy()
    myLib.getCommentsFor("/some/article", callback)
    this.server.respond()

    sinon.assert.calledWith(callback, [{ id: 12, comment: "Hey there" }])
  })
})
```

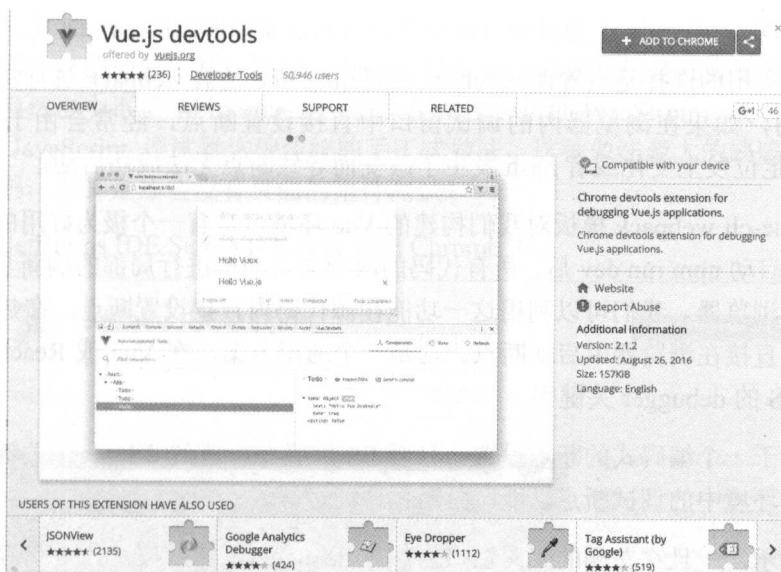
5.4 调试

对于调试相信每一位程序员都不会陌生，作为前端开发者浏览器的调试窗口就更是不可或缺了。

Vue-DevTools

Vue-DevTools 是官方提供的实时调试工具，它是一个 Chrome 的应用插件，可嵌入到 Chrome 的调试器内使用。你需要在 Chrome 网上应用商店内安装 vue-devtools (https://chrome.google.com/webstore/search/vue-devtools?utm_source=chrome-ntp-icon):

只要当前打开的网页有 Vue 实例，这个 Chrome 插件就会自动解释实例结构以及 Vue 实例内的变量，以便我们观测实例运行的情况。



Vue-DevTools 对于初学者来说是一个不错的选择，可以很好地辅助理解 Vue 的运行原理。

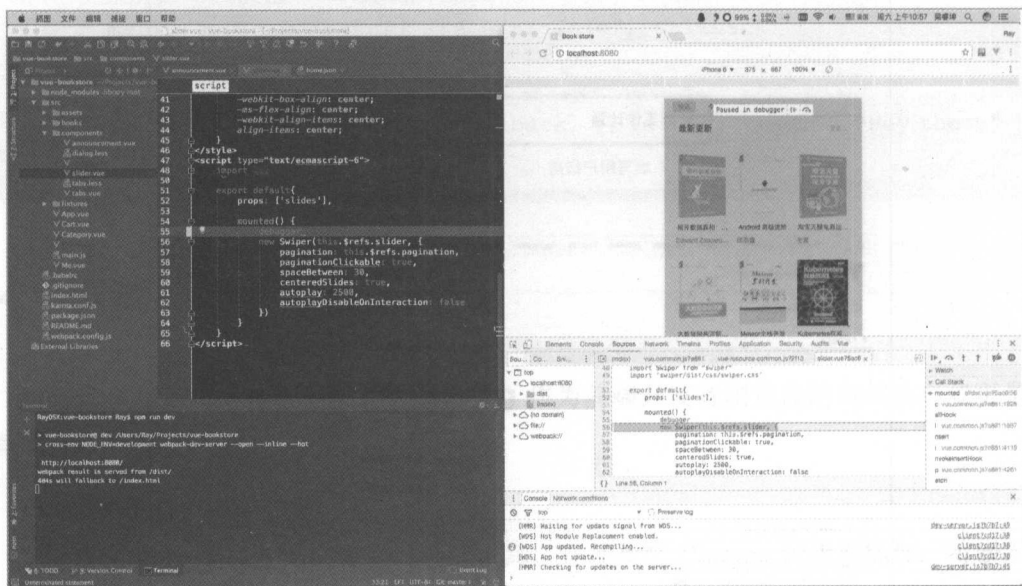
运行期调试

另一个更实用的选择是设置调试的断点，要知道 Vue 是被 webpack 进行实时编译后加载到浏览器的，如果在浏览器内的调试窗口中直接设置断点，经常会由于代码刷新后 Sourcemap 的定位发生变化或者 hash 发生了改变而导致断点无法成功启动。

别忘了 vue-cli webpack 模板对我们构建的 Vue 环境可是有一个极为好用的功能的——“热加载”。当启动 `npm run dev` 后，所有代码的改变会自动地进行局部的刷新与载入，而不需要人工刷新浏览器。我们可以利用这一功能在源代码内直接设置断点，在热加载运行重新载入代码后直接在浏览器中启动断点。这是一个通用方案，在 Vue 或 React 中都可以执行，那就是 ES 的 `debugger` 关键字。

它就相当于一个编码式的断点设置，只要 ES 解释器一遇到 `debugger` 关键字就会自动启用当前宿主环境中的调试断点功能，打断程序的运行。

如下图所示，一旦在左侧的代码窗口内加入 `debugger`，然后按 `Ctrl+S` 保存，右侧的浏览器运行窗口就会自动载入断点并跳转到设置断点的代码上，这是一个极为方便也是常用的开发功能。



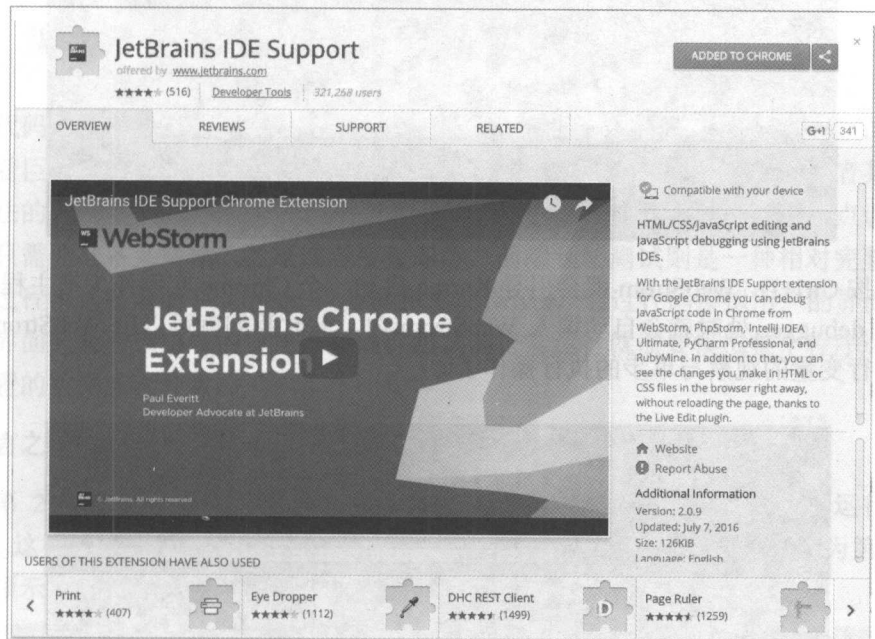
Debugger 在 Chrome 的效果

测试期与 IDE 的集成调试

在浏览器中调试很明显只适用于对界面的精细调整，或者更准确地说是一种手工模式。

但对于我们采用 Karma 测试加载器来执行的全自动化测试就不太适用了。作为专业的开发人员应该使用专业的开发工具，所以 JetBrains 的开发工具集可谓是开发必备的 IDE，我们做前端开发当然也少不了 WebStorm 这一强大的助手。借助 WebStorm，我们摆脱了使用浏览器自带的 JavaScript 调试器这种传统的手工式做法，这种做法最大的缺陷是难以正确定位我们的源码，尤其是那些没有页面的组件测试！

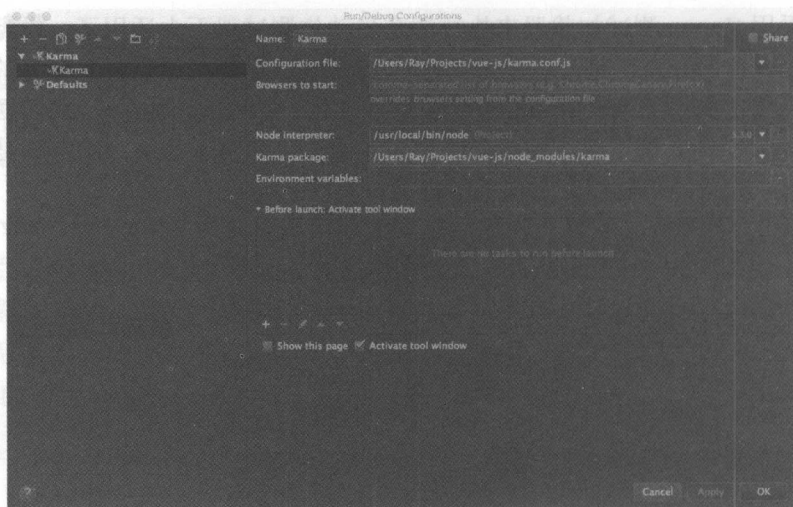
首先将 JetBrains IDE Support 工具安装到 Chrome 中：



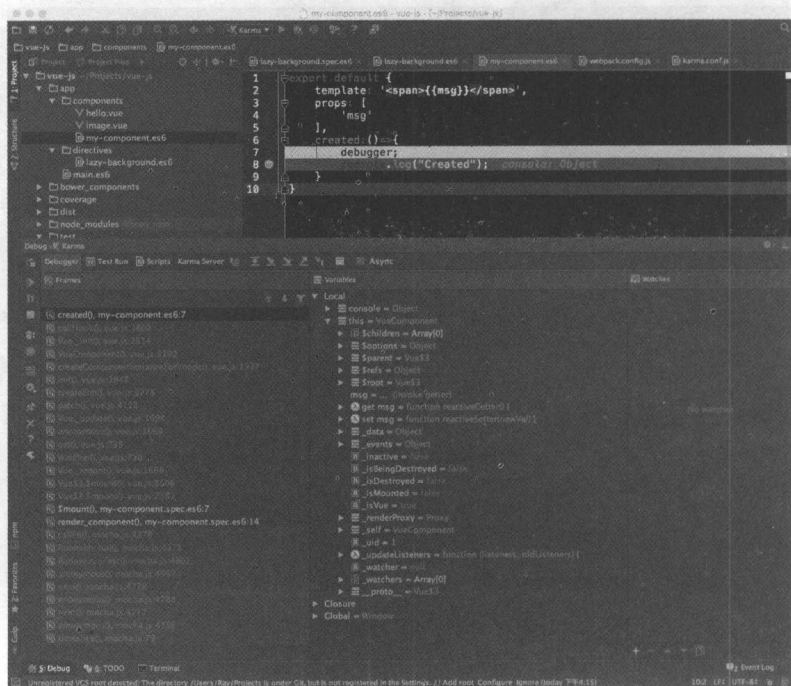
回到 WebStorm，在主菜单中选择“Run/EditConfigurations”选项，弹出以下的配置对话框，点击左上角的+号增加一个 Karma 的配置，然后在 Configuration file 输入框中选择当前项目下的 karma.conf.js，如下图所示。

然后在代码中需要添加断点的地方加入 debugger 关键字：

```
export default {
  created () {
    // 断点
    debugger
    console.log("Created")
  }
}
```

然后按 **Ctrl+D**, WebStrom 就会引导 Karma 启动一个 Chrome 实例作为宿主程序, 当程序执行到 debugger 处时就会自动调入 WebStrom 内, 此时我们就可以用 WebStrom 自带的调试器进行变量的观察与单步的执行操作了。



这个方法无论是 Vue1.0 还是 Vue2.0 都可用。

5.5 Nightwatch 入门

Nightwatch 的开发非常容易学，具体的应用我会在下一章中详细地嵌入到示例中。为了下文描述的方便，我们需要普及一些 Nightwatch 简单的背景知识。如果你已经完全掌握了 Nightwatch 的开发，那么可以跳过以下这一部分。

5.5.1 编写端到端测试

从代码结构上说，端到端测试与单元测试的写法是基本一致的，但在测试的设计思路上却有着巨大的差别。单元测试标注的是局部的代码，即对某个（些）类或者某个（些）具体方法的调用方式及其输出结果进行测试，必要时可以挂入调试器进行断点调试，运行测试前只需要准备某些输入数据或者变量即可。而端到端测试则是一种相对完整的外部操作模拟过程，它要借助 Selenium 服务器和 WebDriver，通过代码模拟用户的界面操作，然后检测界面元素应该出现的变化，要确保测试的正常运行就需要模拟整个程序运行时所有需要配置的数据或者参数。

简言之，端到端测试侧重于检测界面的交互效果与操作逻辑是否正确。

在第 2 章工程化 JS 开发中已经介绍过如何将 Nightwatch 的默认测试运行器配置为 Mocha，这样一来端到端测试的结构就可以与单元测试的写法一致，以下就为第 1 章中的待办事项示例来写一个简单的端到端测试：

```
describe('TODO 界面测试', () => {
  it('应该正确 TODO 界面', (client, done) => {
    client.url('http://localhost:8080')
      .waitForElementVisible('body', 1000)
      .assert.elementPresent('input[type="text"]')
      .getAttribute('input[type="text"]', 'placeholder', result => {
        this.assert.equal(result.value, '快写下您要记住的事吧')
      })
      .end()
  })
})
```

这个测试的作用是运行并在浏览器加载待办事项示例，加载完成后检测是否具有一个带有“快写下您要记住的事吧”提示的文本输入框。端到端测试完全取代了我们用眼睛判断界面是否输出正确这一动作，也就是说，只要将所有人工检测的过程转化为端到端测试，那么我们就有了一套对业务逻辑进行全自动化检测的机制！

作为一名前端开发者，进行运行测试是最平常不过的事了，这个过程大约就是执行以下的步骤：

- (1) 打开浏览器查看运行界面。
- (2) 输入仿真数据（大多数开发者通常随便敲入一些所谓的测试数据）。
- (3) 用眼睛查看输出结果，判断界面是否正确。

相信没有多少开发人员是喜欢做上述这些工作的！不少软件企业雇佣一些刚入门的菜鸟们来做这种测试，企图保证程序的业务正确性，这往往是事与愿违的。只有通过端到端测试取代人们最讨厌做的重复性工作，预先输入最正确的仿真数据才是确保业务逻辑能正确运行的关键！

Nightwatch 使用一个浏览器仿真对象（上述代码中的 `client`）的 `url` 函数打开开发服务器地址，此时开发服务器会自动引导 `webpack` 进行编译输出，`waitForElementVisible` 这个函数将等待浏览器加载完成，这个过程是相当缓慢的。为了避免等待超时可以用 `retry(2)` 函数进行保护，或者将 `waitForElementVisible` 的第二个等待参数的时长增加到 5 秒左右，加载完成后用 `Nightwatch` 的断言工具对目标元素的属性或者文字内容进行检验。

XPath 与 CSS 选择器

由于使用断言判断都是针对单个元素进行的，`Nightwatch` 提供的方法的第一个参数一般都是一个选择器参数，如 `assert.elementPresent('input[type="text"]')`，这个选择器可以是 `XPath` 选择器也可以是 `CSS` 类选择器，默认情况下采用 `CSS` 选择器作为元素定位的方法。如果要切换为 `XPath` 的方式对元素进行定位，可以先调用 `useXpath()` 函数，使用 `useCss()` 就可以重新切换为 `CSS` 选择器，具体做法如以下代码所示。

如果要将 `XPath` 作为默认选择器，可以在配置文件内将 `use_xpath` 设置为 `true`。

```
this.todoDemoTest = browser => {
  browser
    .useXpath() // 使用 XPath 选择器
    .click("//tr[@data-search]/span[text()='Search Text']")
    .useCss() // 切换回 CSS 选择器
    .setValue('input[type=text]', 'Vue')
}
```

BDD 式断言

`Nightwatch` 在 v0.7 版本后加入了 `BDD`（行为式驱动）式的断言库，我们能采用 `expect` 语法来使用代码断言：

```

describe('图书视图', () => {
  it('快速搜索', client => {
    client
      .url('http://localhost:8080')
      .pause(1000)

    // 期待元素将在 1 秒内显示
    client.expect.element('body').to.be.present.before(1000)

    // 期待元素#app 具有 display 的样式类
    client.expect.element('#app').to.have.css('display')

    // 期待元素具有 class 属性并且包含有 示例 文字
    client.expect.element('body').to.have.attribute('class').which.
contains('示例')

    // 期待#searchbox 元素是一个 input 类型的标记
    client.expect.element('#searchbox').to.be.an('input')

    // 期待#searchbox 元素是可见的
    client.expect.element('#searchbox').to.be.visible

    client.end();
  })
})

```

expect 接口提供了一种更加灵活、流畅并且更接近自然语言的方式来定义断言，比原有断言接口有着显著的改进。唯一的缺点是它不能进行链式断言，这样会产生大量的重复性代码。

5.5.2 钩子函数与异步测试

Nightwatch 中可以完全兼容原有 Macha 语法提供的 `before/after` 和 `beforeEach / afterEach` 钩子函数。每个钩子函数都会传入一个 Nightwatch 的浏览器实现参数：

```

describe('测试示例', () => {
  before(browser => {
    console.log('准备...')
  })

  after(browser => {
    console.log('清理...')
  })
})

```

```

    })

    beforeEach(browser => {

    })

    afterEach(() => {

    })

    it('第一步', browser => {
      browser
      // ...
    })

    it('第二步', browser => {
      browser
      // ...
      .end()
    })
  })
})

```

在上面的例子中，方法调用的顺序如下：`before()`→`beforeEach()`→“第一步”→`afterEach()`→`beforeEach()`→“第二步”→`afterEach()`→`after()`。

为了增加向后兼容性，`afterEach` 钩子内是不会传有 `browser` 实例参数的，只有异步钩子 `afterEach(browser,done)` 内才会传入该参数。

所有的钩子及测试函数都具有与 Mocha 一样的异步调用能力，每个函数的第二个参数 `done` 作为异步调用结束的回调函数。

进行异步调用时切记一定要调用 `done` 通知 Nightwatch 完成调用，否则会导致测试执行超时。

```

describe('示例', () => {
  beforeEach((browser, done) => {
    // 执行异步操作
    setTimeout(()=>{
      // 完成异步任务
      done()
    }, 100)
  })

  afterEach((browser, done) => {

```



```
// 执行异步操作
setTimeout(() => {
  // 完成异步任务
  done()
}, 200)
})
})
```

默认情况下 Nightwatch 会将超时控制在 10 秒内（测试单元为 2 秒）。在某些情况下，这可能不足以避免超时错误，可以通过在外部全局变量中定义一个 `asyncHookTimeout` 属性（以毫秒为单位）来增加超时量。

另外，我们可以在钩子函数中向 `done` 函数传入 `Error` 对象通知 Nightwatch 捕获到不明确的错误：

```
describe('示例', () => {
  afterEach((browser, done) => {
    performAsync(err=>{
      if (err) {
        done(err)
      }
      // ...
    })
  })
})
})
```

5.5.3 全局模块与 Nightwatch 的调试

我在最初使用 `vue-cli` 脚手架来创建项目时遇到一个很大的困惑，就是 Nightwatch 无法在 Webstorm 中调试！多次翻阅 Nightwatch 的官方文档，发现官方有一篇专门的文章讲述如何在 Webstorm 中启动 Nightwatch 的调试 (<https://github.com/nightwatchjs/nightwatch/wiki/Debugging-Nightwatch-tests-in-WebStorm>)，可惜的是按照 Nightwatch 提供的方法我一直无法成功开启 Nightwatch 的调试模式，这也意味着 Nightwatch 在 Vue 项目中变得极为鸡肋。经过反复的实验，最后我才发现了这并不是 Nightwatch 不能开启调试，而是 `vue-cli` 初始化的 Nightwatch 存在问题！

`vue-cli` 会为我们创建一个 `nightwatch.conf.js` 和一个 `runner.js` 文件，而问题就出在 `runner.js` 文件中，打开这个文件：

```
// 1. start the dev server using production config
process.env.NODE_ENV = 'testing'
```

```

var server = require('.././build/dev-server.js')

// 2. run the nightwatch test suite against it
// to run in additional browsers:
// 1. add an entry in test/e2e/nightwatch.conf.json under "test_settings"
// 2. add it to the --env flag below
// or override the environment flag, for example: `npm run e2e -- --env
chrome,firefox`
// For more information on Nightwatch's config file, see
// http://nightwatchjs.org/guide#settings-file
var opts = process.argv.slice(2)

if (opts.indexOf('--config') === -1) {
  opts = opts.concat(['--config', 'test/e2e/nightwatch.conf.js'])
}
if (opts.indexOf('--env') === -1) {
  opts = opts.concat(['--env', 'phantom'])
}

var spawn = require('cross-spawn')
var runner = spawn('./node_modules/.bin/nightwatch', opts, {stdio:
'inherit'})

runner.on('exit', function (code) {
  server.close()
  process.exit(code)
})

runner.on('error', function (err) {
  server.close()
  throw err
})

```

会发现这个 **runner** 实际上是用子进程加载 **Nightwatch** 和启动开发服务器，以避免开发服务器启动后将线程独占。而 **WebStorm** 的调试器只能嵌入到启动的主进程中，也就是说，如果用 **debug** 模式来运行 **runner** 的话，调试器就只能进入到开发服务器 **dev-server.js** 引导的后端模拟程序内，而被编译后的 **Vue** 程序将失去调试的机会！这是问题的症结所在。

也就是说，如果要启动 **Nightwatch** 就必须用 **Nightwatch** 作为主进程来引导所有的测试文件，而不能是开发服务器。幸好，我们并不需要去独立编写一个 **runner** 来引导 **Nightwatch**，因为 **Nightwatch** 有另一个机制让我们在钩子里去执行开发服务器的启动这一异步性的操作。

大多数时候，在 **globals_path** 的配置属性中指定一个外部文件定义全局变量会比在

nightwatch.json 中定义更好。你可以根据需要针对不同的测试环境定义全局变量。例如在本地运行的测试和针对远程生产服务器上运行的测试可能存在着不同的全局变量配置策略 (<http://nightwatchjs.org/>)。

以上是我在 Nightwatch 上找到的一段关于“全局钩子”的解析，官网上还提供了一个示例。可能本人比较愚钝，看完他们提供的这个解释和示例源代码后还是觉得一头雾水，根本不知所云，只是凭着一个老程序员的直觉感到这可能是解决异步服务启动的一个办法。实践才是检验真理的唯一标准，动手开干！果然，这个全局钩子真的是一个可以进行异步配置的解决方案。

Nightwatch 将其分为两个部分：“全局钩子”与“全局配置”，但实质上就是一个东西。Nightwatch 允许声明一个全局的模块文件，通过 `globals_path` 配置项引入到 `nightwatch.conf.js` 内，这个模块文件可以重写 `nightwatch` 配置文件中的内容，最重要的是它可以提供全局性的 `begin`、`beginEach`、`after` 和 `afterEach` 这 4 个重要的钩子函数。这意味着在执行所有的 E2E 测试文件之前我们可以引导开发服务器启动，执行 `webpack` 编译这一系列的动作而不至于导致线程的死锁，因为上文已经提到过钩子函数是异步的！

另外，全局配置模块文件的存在意义有点像我们的环境配置，有了全局配置模块，我们可以根据不同的环境策略来配置不同运行环境下的配置参数。例如，在生产环境下我们完全不需要启动开发服务，而是直接连接到生产服务器就可以了。

利用全局模块这一强大的配置功能，只需要在 `nightwatch.conf.js` 中加入以下声明：

```
module.exports = {
  "globals_path": "test/e2e/globalsModule.js",
  "selenium": {
    // ... 省略
  },
  // ... 省略
}
```

然后创建一个 `globalsModule.js` 文件，并加入以下的代码：

```
process.env.NODE_ENV = 'testing'
const server = require('.././build/dev-server.js')
const config = require('.././config');

module.exports = {
  before: function (done) {
```

```

server.listen(config.dev.port, function (err) {
  if (err) {
    console.log(err);
    done(err);
  } else {
    console.log('开发服务器启动侦听...');
    done();
  }
})
},
after: function (done) {
  server.close();
  done();
}
}
}

```

这样 Nightwatch 就会引导开发服务器启动。最后还得对 dev-server.js 做一个小小的修改，因为 dev-server.js 一旦被引入就会自动启动，我们需要对此进行调整，如果是“testing”环境就只导出 express 服务对象的实例而不是返回 express 应用对象的侦听方法的返回值：

```

// build/dev-server.js
// ... 省略
module.exports = process.env.NODE_ENV !== 'testing' ? app.listen(port,
function (err) {
  if (err) {
    console.log(err)
    return
  }
  var uri = 'http://localhost:' + port
  console.log('Listening at ' + uri + '\n')
  opn(uri)
}) : app

```

接下来只要按照 Nightwatch 的官方文档在 WebStorm 中配置运行器就可以了，具体做法如下所示。

- (1) 在“Run”菜单内点击“Edit Configurations...”。
- (2) 创建一个 Node.js 的运行配置项。
- (3) 在“JavaScript file”内填入 node_modules/nightwatch/bin/nightwatch。

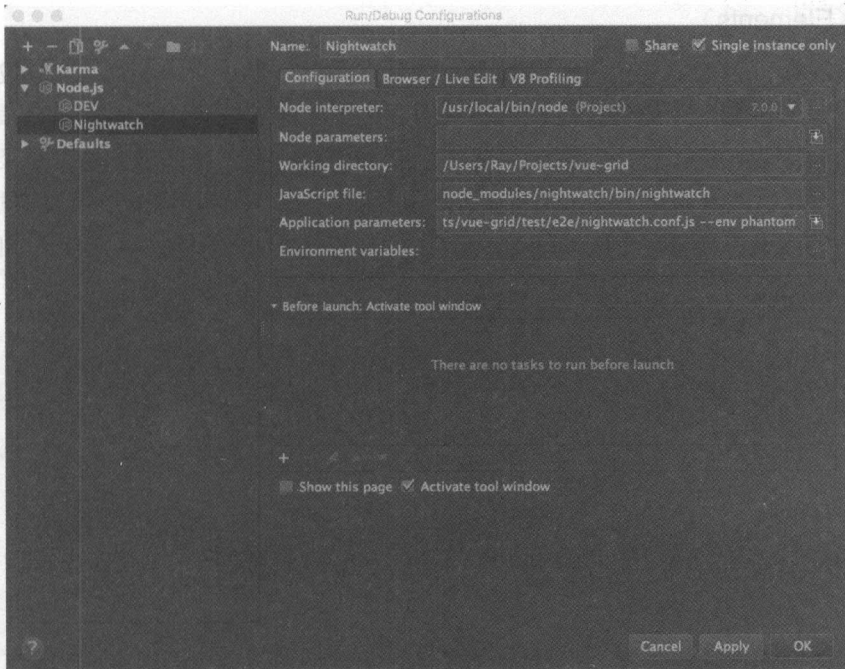
(4) 在“Application parameters”内填入 --config test/e2e/nightwatch.conf.js --env phantom。

(5) 点击“OK”保存。

在测试文件内直接点击代码行断点（无须 `debugger` 关键字），运行“Run→Debug Nightwatch”就可以启动 Nightwatch 的调试模式了。

如果要在命令行运行 E2E 测试，可以在项目的根目录执行以下的语句：

```
$ nightwatch --config test/e2e/nightwatch.conf.js --env phantom
```



5.5.4 Page Objects 模式

Page Objects 模式是由软件大师 Martin Fowler 在 2013 年提出的一种专门用于端到端测试的设计模式 (<https://martinfowler.com/bliki/PageObject.html>)。Page Objects 模式是通过将 Web 应用程序的页面或页面片段包装成一个可实例化的对象以供端到端测试调用的一种模式。它的目的是通过 Page 对象将端到端测试中大量用于查找、定位元素的操作抽象并封装为一些方法，避免由于页面的变更而导致大量代码逻辑分散性地发生变化。

当我们试图测试一个 Web 页面时，不得不依赖页面上的元素去进行交互并确认程序应用是否正常运行。然而当你的脚本试图直接操作页面上的 HTML 元素时，一旦有相关

UI 的变更，那测试将会变得十分脆弱。Page object 模式就是对 HTML 页面以及元素细节的封装，并对外提供应用级别的 API，使你摆脱与 HTML 的纠缠。

——Martin Fowler

Page Objects 模式已被广泛应用在各种主流的端到端测试框架内，当然包括 Nightwatch。

Nightwatch 对 Page Objects 的支持做得相当之好，可以说是与测试实例已经融为一体了！

元素（Elements）

大多数时候，我们需要定义一些元素，并且在测试中通过命令和断言与之交互。如果我们将元素定义放在一个地方，这样就有利于我们集中维护或使用元素的具体属性，特别是在较大的集成测试中，使用元素对象将大大有助于保持测试代码的整洁。需要指出的一点是，这里的元素并不是指 DOM 元素，这是由 Nightwatch 抽象出来的一个元素概念，一个 Page 对象内的元素可以是单个的 DOM 元素，也可以是由多个 DOM 元素复合而成的结合体。

例如，可以用 XPath 和 CSS 混合式地定义元素中的成员：

```
module.exports = {
  elements: {
    searchBar: {
      selector: 'input[type=text]'
    },
    submit: {
      selector: '//[@name="q"]',
      locateStrategy: 'xpath'
    }
  }
};
```

或者，可以更简单地只用 CSS 选择器来定义：

```
module.exports = {
  elements: {
    searchBar: 'input[type=text]'
  }
};
```

不错，Page 对象的元素实际上就是一个选择器而已！

还可以定义一个返回 DOM 集合的选择器元素：

```
var sharedElements = {
  mailLink: 'a[href*="ray@domain.com"]'
```

```
};

module.exports = {
  elements: [
    sharedElements,
    {
      searchBar: 'input[type=text]'
    }
  ]
};
```

每个页面当然要有一个 URL 地址：

```
module.exports = {
  url: 'http://localhost:8080/search',
  elements: {
    searchBar: {
      selector: 'input[type=text]'
    },
    submit: {
      selector: '//[@name="q"]',
      locateStrategy: 'xpath'
    }
  }
};
```

在端到端测试中就可以这样来使用它：

```
describe('图书视图', () => {
  it('快速搜索', client => {
    const searcher = client.page.searcher()
    searcher.navigate()
      .assert.title('图书视图')
      .assert.visible('@searchBar')
      .setValue('@searchBar', 'Vue')
      .click('@submit')
      .end()
  })
})
```

所有的 Page 对象通过 `nightwatch.conf.js` 内的 `page_objects_path` 配置项指定并由 Nightwatch 在运行时自动加载并注入到 `client.page` 属性内，所以我们无须用 `import` 去导入它们，只要放在一个统一的文件夹内就可以了。

分段 (Sections)

很多时候将一个页面定义为多个分段是非常有用的归类手法, 分段主要负责两项工作:

- (1) 为页面划分出一个层级式的“命名空间”。
- (2) 从逻辑上将抽象的元素分布于一个新的树状的对象结构内。

例如:

```
export default {
  sections: {
    menu: {
      selector: '#gb',
      elements: {
        mail: {
          selector: 'a[href="mail"]'
        },
        images: {
          selector: 'a[href="imgghp"]'
        }
      }
    }
  }
}
```

在测试文件中会这样使用它:

```
describe('图书视图', () => {
  it('快速搜索', client => {
    const searcher = client.page.searcher()
    searcher.navigate()
    searcher.expect.section('@menu').to.be.visible
    const menuSection = searcher.section.menu
    menuSection.expect.element('@mail').to.be.visible
    menuSection.expect.element('@images').to.be.visible
    menuSection.click('@mail')
    client.end()
  })
})
```

需要注意的是, 分段对象上的命令与断言将会返回分段对象本身, 用作链式调用。有需要的话, 还可以在分段内定义更小的分段用于分解复杂的页面:

```
export default {
  sections: {
```

```

app: {
  selector: '#app',
  elements: {
    searchbox: {
      selector: 'a[href="mail"]'
    }
  },
  sections: {
    view: {
      selector: 'div.dataview',
      elements: {
        headers: {
          selector: 'th'
        },
        rows: {
          selector: 'tbody>tr'
        },
        newbook_button: {
          selector: 'button.uk-button'
        }
      }
    }
  }
}

```

在测试文件中引用嵌套式分段对象：

```

describe('图书视图', () => {
  it('', client => {
    var books = client.page.books()
    books.expect.section('@app').to.be.visible

    var appSection = books.section.app
    var viewSection = appSection.section.view
    viewSection.click('@newbook_button')

    viewSection.expect.element('@rows').to.have.lengthOf(20)
    viewSection.expect.element('@headers').to.have.lengthOf(5)
    client.end()
  })
})

```


命令 (Commands)

Nightwatch 的命令概念实质上是页面对象上的链式方法，它可以有效地封装与页面相关的逻辑行为，每个命令方法执行完成后都将返回页面对象、分段或者元素本身。

命令只是与普通的 JavaScript 对象定义有点差别，通过 `commands` 属性指定到页面对象内：

```
const bookCommands = {
  search: () => {
    this.api.pause(1000)
    return this.waitForElementVisible('@searchButton', 1000)
      .click('@searchButton')
  }
}

module.exports = {
  commands: [bookCommands],
  elements: {
    searchBar: {
      selector: 'input[type=text]'
    },
    searchButton: {
      selector: '#go_search'
    }
  }
};
```

那么测试代码中的使用示例就应该为：

```
describe('图书视图', () => {
  it('快速搜索', (client) => {
    const books = client.page.books()
    books.setValue('@searchBar', 'Vue')
    .search()
    client.end()
  })
})
```


第 6 章 视图与表单的处理

本章将围绕如何使用 Vue 实现一个对图书资料维护功能的示例展开讲解。我从 2005 年开始进入互联网开发的领域，用过很多种不同的语言，开发过许许多多的互联网应用，在开发这些项目或产品的过程中，表单与视图的处理其实是最多的。甚至可以说，只要涉及数据操作的功能都能被划分到表单处理与视图处理的范围之内。

首先，对**表单**与**视图**这两个我们最常用的逻辑概念和它们自身所发挥的作用进行定义。

视图

用于处理多行的数据集，所以它通常会以列表和表格的方式呈现。正如其名字一样，它只是从不同角度、维度查看一个或多个数据表的一种界面组件。

视图的常规操作有：

- 数据分页——对于数据量很大的数据表我们会将其分成很多个数据页显示，在移动端会表现为以滑动加载的方式渐入分页；
- 条件查询——包括快速查询或者多个条件组合性的查询，用于过滤和筛选目标数据；
- 排序——对各个列进行正向或逆向的数据排序；
- 多行选定——当我们需要对一多行数据进行同一个操作时就需要视图能支持多行选定功能，例如批量删除；
- 添加/编辑/显示单行数据的入口——这是数据视图的一个很重要的功能，即使是一个只读视图我们也应该提供一个能查看数据的详情表单。

视图设计的成功关键是：**只呈现最少量的数据字段与数据行**。视图是一种信息量很大的页面，很多程序员都喜欢将所有的数据列都显示到一个视图当中，甚至将数据行显示得超过了屏幕的高度，每次必须拖动屏幕才能将数据行显示完整。这是一种相当差的使用体验！只要我们站在使用者的角度来思考一下就能体会到：用户通常只关心视图内的“某些数据”，视图只是一个“找”数据的集中地而已，找到他们需要的数据后用户自然会点击详情来了解更多的内容。也就是说，一个视图只需要提供足够的线索让用户快速找到数据就够了。因此，视图最重要的是“**突出重点，快速定位**”。

视图不属于 CRUD 中的任何一个操作，严格点来命名的话它属于 Query，是进行 CRUD 操作的一个极为必要的入口。

表单

在 HTML 中表单就是 form，每个 form 必然会对应一个 action（操作），所以 CRUD 可以看作表单的四种常规行为。

CRUD 就是 Create（创建）、Read（读取）、Update（更新）和 Delete（删除）。删除操作在界面上呈现出来的只是点一下按钮，然后出现一个删除提示，确认后就被执行的一种隐性的界面行为，所以我们可以不将其纳入到表单处理之内。而 CRU 这三个操作刚好能对应三种表单：

- C——空白表单，用于增加数据项；
- R——详情表单，用于显示只读数据，通常用于前台界面；
- U——编辑表单，用于修改数据项。

一个表单设计得是否好用取决于它提供了什么样的输入方式，简单点说就是尽量让用户的输入变得简单，纠正各种可能出现的错误。表单也是使用组件最多、组件结构最复杂的地方，因此以 CRUD 作为 Vue 的组件化示例将是非常有代表性的。

只要对表单与视图这两种基本的“大组件”抽象概念进行分析，配合 Vue 强大的组件化能力，在前端项目开发中你将会有一种如鱼得水的感觉。

6.1 为 Vue2 集成 UIKit

Vue 只是为我们提供了一个很优秀的前端组件式开发框架，但从前面的例子我们都已经了解到，单纯依靠 Vue 是做不出一个漂亮的网页应用的，甚至连“不难看”这个标准都达不到（毕竟它只是一个组件框架），我们总是离不开那种耗费时间的 CSS 或者 Less 的样式表制作过程。在实际开发中，还有很多常用组件，例如，分页、按钮、输入框、导航栏、日期/时间选择器、图片输入，等等。很明显的是这些组件的通用性已不仅仅存在于一个项目内，而是所有的项目都需要！这是个比拼开发速度的年代，我们已经没有时间重复发明轮子了，最正确的选择是使用界面框架，例如 Bootstrap、UIKit、Foundation 等来代替这种大量的重复性极强的界面样式开发工作。

UIKit

Bootstrap 已经有很多年历史了，在业界的应用也相当普遍，无论是前端开发或者后端开发，为了能快速做一个不算太难看的界面，它自然成为众多工程师的选择，包括我。多年下来，Bootstrap 的改进实在是太缓慢了。不客气地说，它基本上就没让我们这些用户感觉它改进过，同质化严重，功能性组件一直不见增加，等等，都让我们只能是痛并着用着。

UIKit 给我们带来了福音，无论从界面上的样式，还是实用组件的数目，甚至到易用性来说都要比 Bootstrap 好上一个层次。唯一的缺陷是它出生得比较晚，可选的主题样式资源不多，毕竟还需要时间让第三方社区来推动发展。但用它来做一个漂亮的交互性强的应用绝对是一个最佳的推荐方案。

Vue 社区上也有一些包装 UIKit 的库，如 vuikit，但它的文档实在太少了，甚至从一开始的安装配套都做得非常差，基本上是脱离了 UIKit 的核心样式包和核心脚本编写的。虽然努力可嘉，但这种功能性复制的包建议还是不要用，前端最耗不起的就是编译包的大小。每个引入的第三方包我们都得吝啬地测算一下得失，即使 webpack 可以用 chunk 来分包，但也不能滥用，否则加载速度缓慢就是破坏使用体验的最大因素。

安装

虽然在 AngularJS、React 和 Vue 的项目中 jQuery 从来都是一个不受欢迎的库。首先是它编译出来后就非常大，而且影响我们的 MVVM 思维，容易因为图方便而又回到 jQuery 那种直接操控 DOM 的死路上去。但 jQuery 的强大在于它的普及性，几乎我们能找到的很多优秀小组件都会有 jQuery 版本，甚至只有 jQuery 的版本。而 UIKit 正是其中一员，不能抗拒的话也只能学会享受。我们得同时安装 jQuery、UIKit 两个库：

```
$ npm i jquery uikit -D
```

配置

我们需要将 jQuery 和 UIKit 的引用以及一些字体的引用配置添加到 webpack 中（UIKit 内置引用了 Fontawesome 字体库），确保已安装了 url-loader 这个库，如果没有安装的话用以下指令进行安装：

```
$ npm i url-loader --D
```

在 webpack.config.js 的 module.rules 配置中加入字体引用配置：

```
rules: [  
  // ... 省略  
  {  
    test: /\.woff2?|eot|ttf|otf|(\?\.*)?$/ ,
```

```

    loader: 'url',
    query: {
      limit: 10000,
      name: '[name].[hash:7].[ext]'
    }
  }
]

```

当然，如果你采用 `vue-cli webpack` 模板来构造项目的话，可以跳过以上的配置。

UIkit 的运行主要依赖于一个主样式文件 `uikit.css`、一个主题文件 `uikit.almost-flat.css`（主题文件内置有三个可选项）和一个脚本文件 `uikit.js`。使用 UIkit 时，需要在代码中同时 `import` 它们才能让 `webpack` 在编译时正确地引用。界面包都是全局性的，那么可以选择在 `main.js` 文件一开始加入引用：

```

import 'jquery'
import 'uikit'
import 'uikit/dist/css/uikit.almost-flat.css'

```

这样写就违反了在第 2 章工程化 `Vue.js` 开发中的一个配置约定，我们不应该将“库”或“依赖包”以全路径方式引入到代码文件中，而应该用 `webpack` 的 `resolve` 配置项，用别名来代替全路径。以下是在 `webpack` 中配置 UIkit 的样式引用别名：

```

resolve: {
  alias: {
    'vue$': 'vue/dist/vue',
    'uikit-css$': 'uikit/dist/css/uikit.almost-flat.css'
  }
}

```

在 `main.js` 代码内引入 UIkit，代码就变为：

```

import 'jquery'
import 'uikit'
import "uikit-css"

```

制作 UIkit 的 `Vue` 插件

上述的写法还是不够 `DRY`，为了使用一个包就得引入多个不同的依赖库，这种做法实在很难看，此时我们可以选择一个 `Vue` 的最佳做法，就是用插件形式来包装这种零碎化的引入方式。在 `src` 根目录下新建一个 `uikit.js` 的文件，然后用 `Vue` 的插件格式来进行包装。以下代码中直接向 `Vue` 实例注入了 UIkit 的一些常用的帮助方法：

```

import 'jquery'
import 'uikit'

```

```
import 'uikit-css'

export default (Vue, options) {

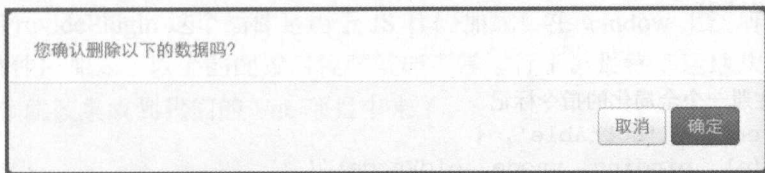
  // 向实例注入 UIKit 的对话框类方法
  Vue.prototype.$ui = {
    alert: UIKit.modal.alert,
    confirm: UIKit.modal.confirm,
    prompt: UIKit.modal.prompt,
    block: UIKit.modal.block
  }
}
```

完成 uikit.js 的编写就可以改写 main.js 的内容了：

```
import UIKit from './uikit'
Vue.use(UIKit)
```

由于对 `Vue.prototype` 进行了扩展，那么就可以像 `vue-resource` 那样在每个 `Vue` 实例内的 `this` 方法中注入一个 `$ui` 对象，用以下方法来显示简单的对话框：

```
methods: {
  delItem() {
    this.$ui.confirm('您确认要删除以下的数据吗？', () => {
      // 这里编写对数据进行删除的代码
    })
  }
}
```



上述的 `confirm` 方法有一个明显的弱点，就是在回调时 `this` 上下文会指向 `window` 而不是 `Vue` 实例本身，这样的话对于编码的使用体验就很差了。我们可以在插件内对 `confirm` 做一个修饰，将回调方法的 `this` 重新指向 `Vue` 实例：

```
Vue.prototype.$ui = {
  // ... 省略
  confirm (question, callback, cancelCallback, options) {
    UIKit.confirm(question,
      callback || callback.apply(this),
      cancelCallback || cancelCallback.apply(this),
```



```
    options)
  }
}
```

`apply` 函数是 ECMA JavaScript 的标准函数，用于更改调用方法上传递的上下文对象。上述代码就是将回调函数的上下文强制替换为当前的 `Vue` 实例，避免了回调上下文丢失而需要手工去定义变量，“hold 住”原有 `this` 上下文的痛苦。

关于 `apply` 函数详细说明可以参考以下链接：https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Function/apply。

现在的代码是不是感觉干净多了？那么回过头来看 `Vue` 的插件，在这里面我们不仅可以像上述代码那样单纯地对 `Vue` 实例进行扩展，还可以进行更多的全局化的处理。当然这里的全局是指这个插件库被引入 `Vue` 并调用 `use` 方法后，例如，我们可以将一些必要的组件或者指令混入插件方法内：

```
export default = (Vue, options) => {

  // 1. 注入全局化的方法
  Vue.myGlobalMethod = () => {
    // ...
  }

  // 2. 进行必要组件的注册
  Vue.component('html-editor', {
    HtmlEditor
  })

  // 3. 注册一个全局化的指令标记
  Vue.directive('sortable', {
    bind (el, binding, vnode, oldVnode) {
      // something logic ...
    }
    ...
  })

  // 4. 注入一些组件的选项
  Vue.mixin({
    created: function () {
      // ...
    }
    ...
  })
}
```

```

    })

    // 5. 扩展实例
    Vue.prototype.$ui = {}

  }

```

UIKit 中的坑

当运行以上的代码后，会很沮丧地发现浏览器中总会出现 UI.\$ 为空的异常，具体显示如下：

```
Type error UI.$ is undefined.
```

我曾尝试过直接跳入 UIKit 的源代码中查找 UI.\$，这个变量其实是对 jQuery 的一个内部引用，准确地说这是在引用 jQuery 的脚本后由 jQuery 注册到浏览器的 window 全局变量上的 jQuery 实例。估计是 UIKit 在生成加载代码时变量的映射与初始化顺序出现问题了。后来想了个办法，直接在 webpack.config.js 配置内对全局变量进行改写，具体代码如下：

```

plugins: [
  new webpack.ProvidePlugin({
    $: "jquery",
    jQuery: "jquery",
    "window.jQuery": "jquery",
    "window.$": "jquery"
  })
]

```

webpack.ProvidePlugin 这个插件是用于 JS 代码加载后在 window 上注册全局变量的一个 webpack 插件，加入了以上的配置后程序就能正常运行了。最终幸运地从大坑中逃出生还！这样 UIKit 就被集成到我们的 Vue 项目中来了。

6.2 表格视图的实现

按照第4章组件化的设计与实现方法中总结的思路，一开始先不要考虑如何去组件化，好代码是重构出来的不是写出来的，所以一开始的冗余反而是让我们找到重构点和组件化起源的地方。回顾组件化的工作流程：

依葫芦画瓢

代码去重

抽取数据结
构

采集与制作
样本数据

分析设计组
件接口

组件内部的
细化与重构

首先是画出功能区块，填入占位符：

```
<tempalte>
  <div id="app">
    <!-- 页头 -->
    <!-- 工具栏 -->
      <!-- 图书统计 -->
      <!-- 搜索框 -->
      <!-- 按钮组 -->
    <!-- 工具栏 -->
    <!-- 页头 -->

    <!-- 正文 -->
    <!-- 图书数据表格 -->
    <!-- 正文 -->
    <!-- 对话框-->
    <!-- 图书编辑/新建 数据表单 -->
    <!-- 对话框-->
  </div>
</tempalte>
```

接下来就是分别将各占位符上的页面模板写出，这个过程我们在之前的章节已经很详细地论述过，此处就不再赘述了，直接上代码：

```
<template>
  <div id="app">
    <!-- 页头 -->
    <div class="uk-block uk-block-primary uk-contrast page-header">
      <div class="uk-container-center">
        <h1 class="uk-heading-large">图书
        <small>Vue CRUD 示例</small>
      </h1>
    </div>
  </div>
  <!-- 页头 -->
  <!-- 页面正文 -->
  <div class="content">
    <!-- 工具栏 -->
    <div class="uk-grid uk-margin-large-bottom">
      <div class="uk-width-3-4">
        <div class="uk-grid">
          <!-- 图书统计 -->
          <div class="uk-width-1-3">
            <span class="uk-text-large uk-text-muted">共有
            <span
```

```

class="uk-text-bold">{{ books.length }}</span>本图书</span>
</div>
<!-- 图书统计 -->
<!-- 搜索框 -->
<div class="uk-width-2-3">
  <div class="uk-form">
    <div class="uk-form-icon">
      <i class="uk-icon-search"></i>
      <input type="search"
        class="search-box
uk-form-width-large"
placeholder="请输入您要筛选的书名
"/></div>
    </div>
  </div>
  <!-- 搜索框 -->
</div>
</div>
<div class="uk-width-1-4">
  <div class="uk-float-right">
    <button title="删除已选中的图书"
      class="uk-button uk-button-danger"
    ><i class="uk-icon-trash"></i>
    </button>

    <button class="uk-button uk-button-primary">
      <i class="uk-icon-plus"></i> <span>添加</span>
    </button>
  </div>
</div>
</div>
<!-- 工具栏 -->
<!-- 图书数据表格 -->
<table class="uk-table uk-table-striped">
  <thead>
    <tr>
      <th class="uk-text-center disable-select">书名</th>
      <th class="uk-text-center uk-width-1-6 disable-select">
类别</th>
      <th class="uk-text-center uk-width-1-6 disable-select">
出版日期</th>
    </tr>
  </thead>
  <tbody>

```



```

        <tr v-for="book in books">
          <td class="book-name uk-form uk-grid">
            <div class="uk-width-1-10">
              <input type="checkbox"
                class="uk-margin-right"/>
            </div>
            <div class="uk-width-9-10">
              <a class="uk-h3"
                href="javascript:void(0)"
                :title="book.name">{{ book.name }}</a>
              <p class="authors" uk-text-muted
                uk-text-small">{{ book.authors }}</p>
            </div>
          </td>
          <td class="small">{{ book.category }}</td>
          <td class="published"
            uk-text-center">{{ book.published }}</td>
        </tr>
      </tbody>
    </table>
    <!-- 图书数据表格 -->
  </div>
  <!-- 页面正文 -->
  <!-- 对话框-->
  <!-- 图书编辑/新建 数据表单 -->
  <!-- 对话框-->
</div>

</template>
<script>
  import "../assets/site.less"

  export default {
    data () {
      return {
        books: [],
      }
    }
  }
</script>

```

然后是提出数据结构:

```

[
  {

```



```

"name": "书名",
"authors": [
  "作者"
],
"editors": [
  ""
],
"series": "电商精英宝典系列",
"isbn": "978-7-121-28410-6",
"published": "2016-04-22",
"pages": 288,
"format": "16(185*235)",
"status": "上市销售",
"en_name": "",
"category": "新经济、互联网思维与电子商务",
"summary": " ",
"price": 79.0
},
// ...
]

```

各字段说明如下表所示。

字 段	类 型	说 明
name	string	书名
authors	Array	作者名列表
editors	Array	编辑名列表
series	string	所属系列
isbn	string	书号
published	string	出版日期
pages	Number	页数
format	string	格式
status	string	发行状态
en_name	string	原名（仅外文书）
category	string	类别（以“、”分隔多个类别名称）
summary	string	全书摘要
price	Number	售价

由于表格的内容是同质化的，所以就不要像之前的示例那样将所有的数据都写成标记了，毕竟示例只是带出一种思路，在我们没有分析出数据结构时要这样做，反之则可以直接进入数据样本的准备阶段。本示例的样本数据比较多就不在这里罗列了，有兴趣的读者

可以到本书的 [github](#) 上查看具体的文件内容。

同样地，我们将数据样本保存到 `~/fixtures/books.json` 文件内，然后直接引入到当前的 `App.vue` 代码内使用：

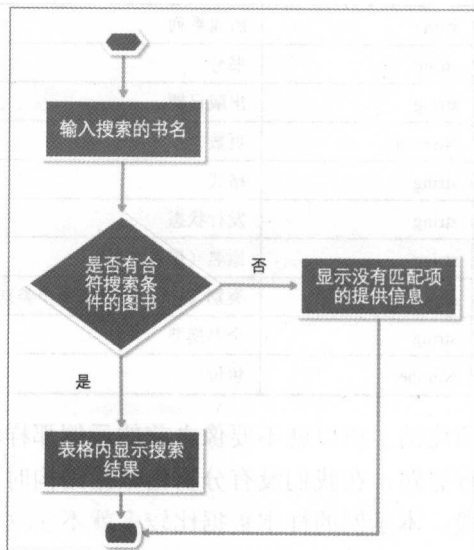
```
import BookData from "../fixtures/books.json"

export default {
  data () {
    return {
      books: BookData
    },
    // ...
  }
}
```

6.2.1 实时数据筛选

界面元素与数据结构的设计与实现都已基本完成，可以说整个程序的轮廓已基本显现。完成外观后就要实现程序的“行为”逻辑了。我们从易到难逐步地实现，首先实现数据的筛选功能。

我们希望在搜索框中一边输入文字，下方的数据行界面就自动按照输入的内容进行筛选，仅显示与搜索框内容相匹配的内容，当没有找到任何数据时显示提示文字“抱歉，没有找到任何的图书数据”，以下是实现的思路流程：



首先，将搜索框 input 的 value 保存到一个 terms 的变量内，由于我们希望界面的刷新会随着这个变量的变化产生改变，那么就应该使用双向绑定的方式，代码如下：

```
<!-- 搜索框 -->
<div class="uk-width-2-3">
  <div class="uk-form">
    <div class="uk-form-icon">
      <i class="uk-icon-search"></i>
      <input type="search"
        v-model="terms"
        class="search-box uk-form-width-large"
        placeholder="请输入您要筛选的书名"/>
    </div>
  </div>
</div>
<!-- 搜索框 -->
```

当 terms 产生变化时，我们得计算出与 terms 相关的搜索结果，上文将数据行的循环绑定到 books 数组上，但是这个 books 数组是原数据，是不应该变化的，那么这里我们就可以用计算属性来进行结果的筛选，并在行循环中将原有的 books 替换掉：

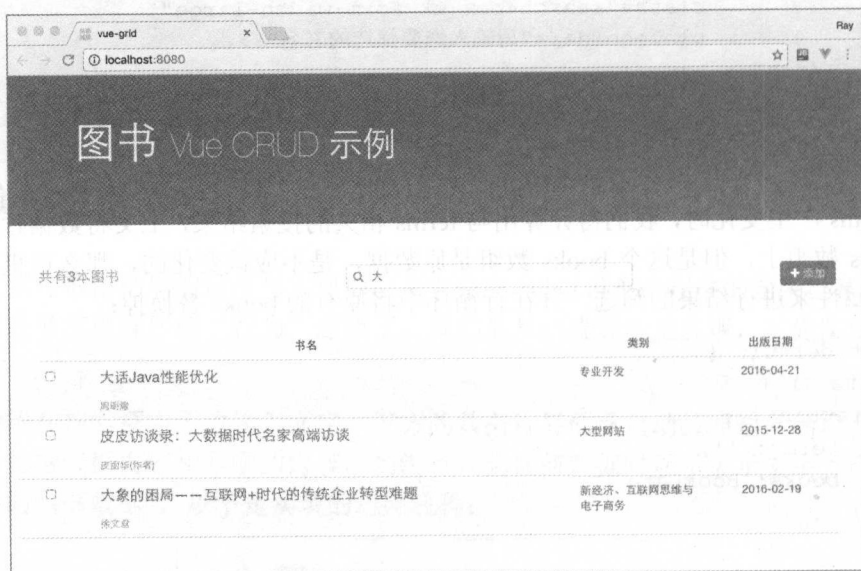
```
export default {
  data () {
    return {
      terms: '',
      books: BookData
    }
  },
  computed: {
    bookFilter () {
      // 用函数式将书名包含有 terms 内容的图书都筛选出来，如果没有则返回原数组
      return this.terms.length ? this.books.filter(x =>
        x.name.indexOf(this.terms) > -1) : this.books
    },
    // ... 省略
  }
}
```

这个 bookFilter 属性一旦被放置于 template 内，只要 terms 发生任何变化，界面都将被重绘，那么将 template 的行循环替换为 bookFilter：

```
<!-- 图书数据表格 -->
<table class="uk-table uk-table-striped" v-if="bookFilter.length">
  <!-- 省略 -->
  <tbody>
```

```
<tr v-for="book in bookFilter">
  <!-- 省略 -->
</tr>
</tbody>
</table>
<div class="uk-text-muted uk-text-large uk-text-center"
  v-if="bookFilter.length==0">抱歉，尚没有找到任何符合条件的图书</div>
```

对 `bookFilter` 的数组长度进行判断，有数据才显示表格，反之则显示提示文字，最终的运行效果如下：



当没有找到数据时应该显示成这样：



接下来就要为程序写 E2E 测试了，创建 `test/e2e/books.spec.js` 文件，按照前文流程图的

逻辑来写 E2E 测试，代码如下所示。

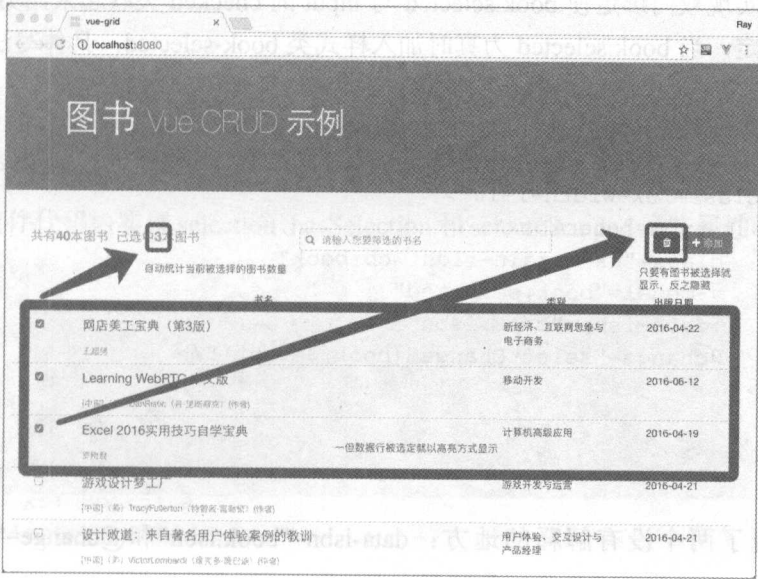
```
describe('图书管理视图', () => {

  it('应该筛选与搜索框输入匹配的图书数据', (client) => {
    const terms = '大数据'
    client.url(client.launchUrl)
    .waitForElementVisible('body', 30000)
    .setValue('input[type="search"]', [terms, client.Keys.ENTER])
    .assert.containsText('.book-name', terms)
    .setValue('input[type="search"]', ['不存在的数据', client.Keys.ENTER])
    .assert.elementPresent('.empty-holder')
    .end()
  })
})
```

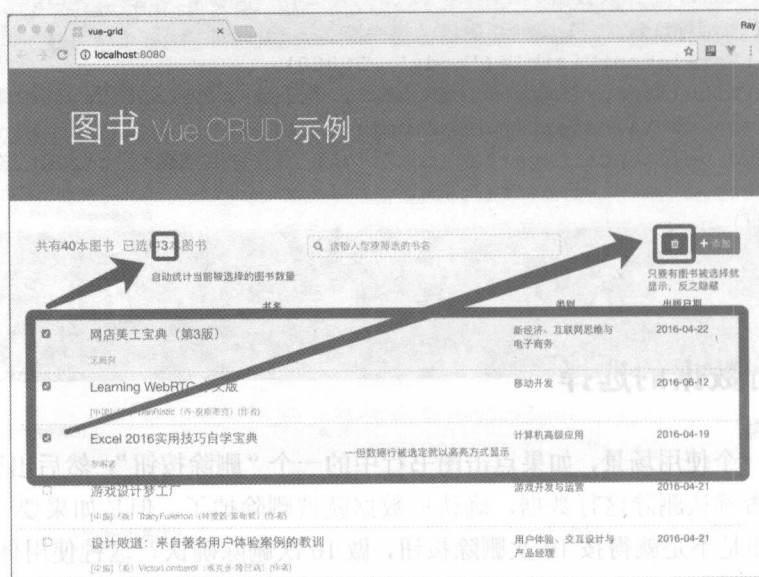
6.2.2 多行数据的选择

想想这样一个使用场景，如果点击图书行中的一个“删除按钮”，然后出现一个提示对话框，询问是否确认删除这行数据，确认后数据就被删除掉了，但是如果一次性删除 10 条图书数据，那是不是就得按 10 次删除按钮，做 10 次删除确认？这种使用体验就太差了，所以需要有多行选择一次性确认删除的功能。

具体的操作效果设计如下图所示。



这里我们可以运用 Vue 的双向绑定和计算属性两个技术点来实现。我们需要为 book 添加一个标识属性，selected 用来记录是否被选择，并将其绑定到 input[type=checkbox]上。虽然这个 selected 是一个无中生有的属性，但这不是问题，因为双向绑定会帮我们处理，没有的话会自动为 book 添加上这一属性，下图是具体的设计思路。



请留意：这里出现了多处共享 book.selected 的状态，此时状态的变更开始变得复杂。

在模板上实现双向绑定使 book.selected 与 input 的 checked 关联起来，另外就是在<tr>上进行属性绑定，当 book.selected 为真时加入样式类 book-selected，具体写法如下：

```
<tr v-for="book in books"
  :class="{ 'book-selected': book.selected}">
  <td class="uk-form uk-grid">
    <div class="uk-width-1-10">
      <input type="checkbox"
        class="uk-margin-right cb-book"
        v-model="book.selected"
        :data-isbn="book.isbn"
        @change="selectChanged(book, $event)" />
    </div>
  </td>
  <!-- 省略 -->
</tr>
```

这里出现了两个没有解释的地方：data-isbn="book.isbn"和@change="selectChanged

(book,\$event)", 这里先卖个关子, 在下文中会解释它们的作用。

在左上方的选择统计标签上加上 `selection.length` 的字面量引用 `selection`(这个变量现在还没有,我们在下方的组件代码内才会补充实现):

```
<!-- 图书统计 -->  
<div class="uk-width-2-4">  
    <span class="uk-text-large uk-text-muted">共有<span  
        class="uk-text-bold">{{ books.length }}</span>本图书  
<span v-if="hasSelection">  
    &nbsp;&nbsp;&nbsp;&已选中<span  
        class="uk-text-bold">{{ selection.length }}</span>本图书  
</span>  
</span>  
</div>
```

最后在删除按钮上加上 v-if 指令进行自动消隐控制:

```
<!-- 按钮组 -->
<div class="uk-width-1-4">
  <div class="uk-float-right">
    <button title="删除已选中的图书"
      class="uk-button uk-button-danger"
      id="btn-delete"
      v-if="hasSelection"
    ><i class="uk-icon-trash"></i>
  </button>
  <!--省略-->
</div>
</div>
```

这里用一个计算属性 `hasSelection` 对 `selection.length` 进行包装，直接在此写上表达式是为了让代码都易读。

最后在组件代码内实现 selection、hasSelection 和 selectChanged 这些属性和事件处理器:

```
// ... 省略
import _ from 'lodash'

export default {
  data () {
    return {
      terms: '',
      books: Bookdata,
      selection: []
    }
  }
}
```

```
    },  
    computed: {  
      hasSelection () {  
        return this.selection.length > 0  
      }  
    },  
    methods: {  
      selectionChanged (book) {  
        if (e.target.checked) {  
          this.selection.push(book.isbn)  
          // 取唯一值  
          this.selection = _.uniq(this.selection)  
        } else {  
          // 排除符合条件的数据并返回新的数组  
          this.selection = _.reject(this.selection, b => book.isbn === b)  
        }  
      },  
      // ... 省略  
    }  
  }  
}
```

这就是 `selectionChanged` 的真相，这个事件处理器是将图书的 `isbn` 保存到 `selection` 数组内，这样做是为下文中对数据进行批量删除时做数据准备的。

如果你没有用过 `underscore` (<http://underscorejs.org>) / `lodash` (<https://lodash.com>)，或者没有接触过函数式编程，可能会对上述代码有所困惑。这里使用了 `lodash` 中的两个高阶函数 `uniq` 和 `reject`，分别对 `selection` 数组进行处理。函数式编程可以极大地提高代码运行效能，大幅度减少代码量，而且代码可读性更强。可能你一时间不明白这两个函数是怎么实现的，但这并不重要，因为函数名已解释了它们自身的用法，这是函数式能自描述 (Self-Describe) 的一种特点。

函数式编程是一个很广泛的内容，它是一种通用的方法论。在本书短短的篇幅内实在无法过多地讨论，但如果你对它有兴趣，那么可以关注我写的另一本书《攀登架构之巅》，在那里深度地了解函数编程的方方面面，另外还有一本非常好的书《Functional JavaScript》[2013 Michel Fogus O'REALY]，这是一本将我引入函数式编程领域的极好的范本。

`underscore` 和 `lodash` 是在 JavaScript 中使用函数式编程的必备高阶函数库，这是两个同质的类库，引用了其中一个另一个就没有存在意义了，`lodash` 会比 `underscore` 更好用一些。

lodash 的安装很简单:

```
$ npm i lodash -D
```

然后如上文一样直接引入使用即可。

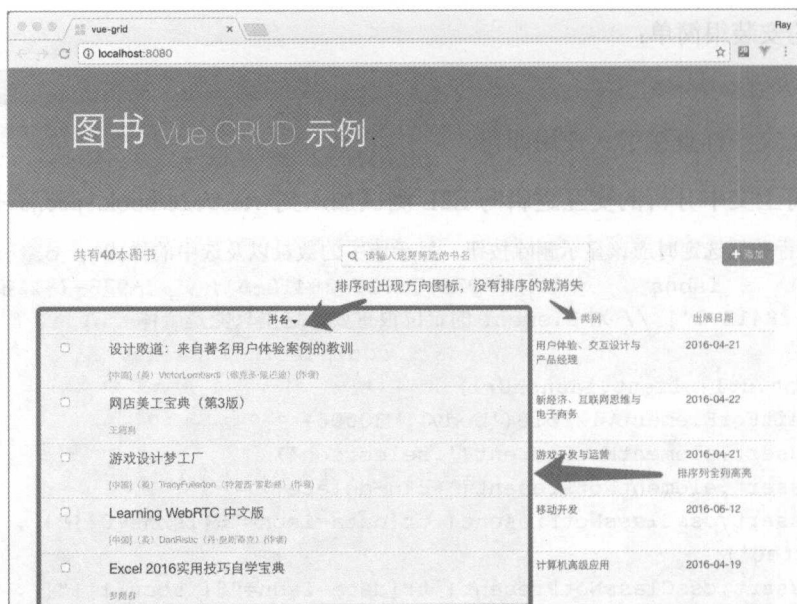
最后, 将上文中分析的交互逻辑写 E2E 测试加入到 `~/test/e2e/book.spec.js` 中:

```
it('多行数据选定时应该显示删除按钮、显示选中的数量以及选中的样式', client => {
  const isbnns = ['978-7-121-28410-6', '978-7-121-28817-3',
    '978-7-121-28413-7'] // 对 Element 的定位很重要, 这里只能是个体

  client.url(client.launchUrl)
    .waitForElementVisible('body', 30000)
    .assert.elementNotPresent('.selection')
    .assert.elementNotPresent('#btn-delete')
    .assert.cssClassNotPresent(`tr[data-isbn="${isbnns[0]}"]`,
      'book-selected')
    .assert.cssClassNotPresent(`tr[data-isbn="${isbnns[1]}"]`,
      'book-selected')
    .assert.cssClassNotPresent(`tr[data-isbn="${isbnns[2]}"]`,
      'book-selected')
    .click(`input[type="checkbox"][data-isbn="${isbnns[0]}"]`)
    .click(`input[type="checkbox"][data-isbn="${isbnns[1]}"]`)
    .click(`input[type="checkbox"][data-isbn="${isbnns[2]}"]`)
    .assert.containsText('.selection', '3')
    .assert.elementPresent('#btn-delete')
    .assert.cssClassPresent(`tr[data-isbn="${isbnns[0]}"]`,
      'book-selected')
    .assert.cssClassPresent(`tr[data-isbn="${isbnns[1]}"]`,
      'book-selected')
    .assert.cssClassPresent(`tr[data-isbn="${isbnns[2]}"]`,
      'book-selected')
    .end()
})
```

6.2.3 排序的实现

接下来就要实现更为复杂的交互效果了, 在这个示例中我希望这个表格能实现像 Excel 一样的排序效果, 具体如下图所示。



首先我们要保持住两个状态：

- `sortingKey`——当前排序的字段名称；
- `direction`——排序的方向。

然后在模板上对表格内容进行重构：

```
<table class="uk-table uk-table-striped">
  <tr>
    <th class="uk-text-center disable-select"
      :class="{ 'sorting': sorted('name') }"
      data-col="name"
      @click="sortBy('name')">
      <div>书名
      <span :class="{
        'uk-icon-sort-asc': direction=='asc',
        'uk-icon-sort-desc': direction=='desc'
      }">
      </span>
    </th>
    <th class="uk-text-center uk-width-1-6 disable-select"
      :class="{ 'sorting': sorted('category') }"
      data-col="category">
```



```

        @click="sortBy('category')">
<div>类别
    <span :class="{
        'uk-icon-sort-asc': direction=='asc',
        'uk-icon-sort-desc': direction=='desc'
    }">
        v-if="sortingKey=='category'"></span></div>
</th>
<th class="uk-text-center uk-width-1-6 disable-select"
    :class="{ 'sorting':sorted('published') }"
    data-col="published"
    @click="sortBy('published')">
<div>出版日期
    <span :class="{
        'uk-icon-sort-asc': direction=='asc',
        'uk-icon-sort-desc': direction=='desc'
    }">
        v-if="sortingKey=='published'"></span></div>
</th>
</tr>
</thead>
<tbody>
<tr v-for="book in bookFilter"
    :class="{ 'book-selected': book.selected }"
    :data-isbn="book.isbn">
    <td class="uk-form">
        <div class="uk-grid"
            :class="{ 'sorting':sorted('name') }">
            <!-- 书名单元 内容省略-->
        </div>
    </td>
    <td class="small">
        <div class="fill"
            :class="{ 'sorting':sorted('category') }">
            {{ book.category }}
        </div>
    </td>
    <td class="published uk-text-center">
        <div class="fill"
            :class="{ 'sorting':sorted('published') }">
            {{ book.published }}
        </div>
    </td>
</tr>

```

```

    </tbody>
  </table>

```

这样一次性地看代码是否有点眼花缭乱？全贴出来是为了方便对照阅读，下面就分开来解释，这样会更清楚其中的逻辑。排序的触发是由列的点击事件引起的，将上面模板的代码抽象化成一种模式的话就会变成以下这样：

```

<th :class="{ 'sorting':sorted('字段名')}"
    @click="sortBy('字段名')">
  <div>书名
    <span :class="{
      'uk-icon-sort-asc': direction=='asc',
      'uk-icon-sort-desc': direction=='desc'
    }"
      v-if="sortingKey=='字段名'"></span></div>
</th>

```

代码虽多但实际逻辑并不复杂，一个是调用排序方法，另一个是进行样式与排序图标的消隐控制。用同样的方法来看数据单元格就更简单了，只是实现了样式的切换：

```

<td>
  <div :class="{ 'sorting':sorted('字段名')}">
    {{ 字面量 }}
  </div>
</td>

```

这样我们需要在组件代码内实现 `sorted(fieldName)` 来判断输入的字段是否正在排序；`sortBy(fieldName)` 是对数据进行排序。

```

export default {
  data () {
    return {
      terms: '',
      sortingKey: '',
      direction: 'asc',
      statusText: '',
      books: BookData,
      selection: []
    }
  },
  methods: {
    sorted (key) {
      return key === this.sortingKey
    },
    sortBy (key) {

```

```

    if (key === this.sortingKey) {
      // 对排序方向进行互斥式交换
      this.direction = this.direction === 'asc' ? 'desc' : 'asc'
    }
    this.sortingKey = key
    this.books = _.orderBy(this.books, key, this.direction)
  },
  // ... 省略
},
// ... 省略
}

```

实际上最终只需要调用 `lodash` 中的 `orderBy` 方法就可以对指定 `key` 和排序方向上的对象数组实现排序。

上文的代码中为每一个 `th` 都加入了一个 `data-col` 属性，这是为 E2E 测试而准备的，以下是排序操作的 E2E 测试代码：

```

it('点击列头时应该进行排序', client => {
  const colName = 'th[data-col="name"]'
  const colCat = 'th[data-col="category"]'
  const colPub = 'th[data-col="published"]'
  const sortingClass = 'sorting'
  const asc = 'div>span.uk-icon-sort-asc'
  const desc = 'div>span.uk-icon-sort-desc'

  client.url(client.launchUrl)
    .waitForElementVisible('body', 30000)
    .assert.cssClassNotPresent(colName, sortingClass)
    .assert.cssClassNotPresent(colCat, sortingClass)
    .assert.cssClassNotPresent(colPub, sortingClass)
    .assert.elementNotPresent(`${colName}>div>span`)
    .assert.elementNotPresent(`${colCat}>div>span`)
    .assert.elementNotPresent(`${colPub}>div>span`)
    .getAttribute('tbody>tr:first', 'data-isbn', result => {
      this.assert.equal(result.value, '978-7-121-28410-6') // 无排序
    })
    .click(colName) // 对名称进行排序
    .assert.elementPresent(`${colName}>${asc}`)
    .getAttribute('tbody>tr:first', 'data-isbn', result => {
      this.assert.equal(result.value, '978-7-121-28413-7') // 升序
    })
    .click(colName) // 反向排序
    .getAttribute('tbody>tr:first', 'data-isbn', result => {

```

```

        this.assert.equal(result.value, '978-7-121-28381-9') //降序
    })
    .assert.elementPresent(`${colName}>${desc}`)
    .assert.cssClassPresent(colName, sortingClass)
    .assert.cssClassNotPresent(colCat, sortingClass)
    .assert.cssClassNotPresent(colPub, sortingClass)
    .click(colCat) // 对类别进行排序
    .assert.elementPresent(`${colCat}>${asc}`)
    .assert.cssClassPresent(colCat, sortingClass)
    .assert.cssClassNotPresent(colName, sortingClass)
    .assert.cssClassNotPresent(colPub, sortingClass)
    .end()
})

```

在写 E2E 测试的时候你才会发现原来的代码中有很多要进行操作的目标元素，通过代码是没有办法定位的，这个时候我们就得向这些元素加入一些特殊的属性或者 CSS 类作为标识。当然，这些辅助属性的命名仍然要按照我们编码前约定的规则，要有可读性，千万不要用拼音或者数字一类让人摸不着头脑的方式命名，否则这将会毁掉你的 E2E 测试。加入了辅助属性，代码才真正完整，因为具有辅助属性的 HTML 结构才基本符合 SEO（搜索引擎优化）要求，可以说这是一种额外的收获吧。

6.3 单一职责原则与高级组件开发方法

到此已经完成了视图的实现，接下来就要实现表单部分的功能。但是，现在的代码已经变得越来越“胖”了，此时视图页中的代码行已经超过 200 多行了，要在源码中找一个组件代码上定义的方法已经越来越不方便了。其实，有这种感觉就对了，最怕的是当我们写到 1000 行或者 3000 行的时候还麻木不仁地在加代码。一个文件代码行最多不超过 100，这是最基本的编码约定，无论何种语言都适用。

从架构设计的角度来看，现在这个 App 在功能上肩负了太多的职责——页面布局、数据获取、排序、CRUD、数据表单，分页、数据筛选，等等——已经严重地违反了“单一职责原则”。这就好像是一个人虽然他很能干，但是如果你什么事都让他来干，他做错事的机率会大大增加。

单一职责原则：不要存在多于一个导致类变更的原因。通俗地说，即一个类只负责一项职责。

现在正是对代码进行全面梳理和重构的时候，我们的代码已经大量充斥着各种 UIKit

的样式与结构，一堆的结构下页只能完成一个小小的功能。按这种趋势发展，代码已经在逐渐进入“意大利面条式代码”（意思是纠缠在一起无法分开）的状态了。我们要将这些啰唆的逻辑全面重构为各个小的组件，每个组件承担起单一的职责，直到不可细分为止。

从一开始页头的代码就能被组件化：

```
<div class="uk-block uk-block-primary uk-contrast page-header">
  <div class="uk-container-center">
    <h1 class="uk-heading-large">图书
    <small>Vue CRUD 示例</small>
  </h1>
</div>
</div>
```

组件化后变成：

```
<page-header header="图书"
  sub-header="Vue CRUD 示例">
</page-header>
```

这个 PageHeader 组件实现很简单，就是 header 和 sub-header 两个输入参数，具体的代码如下：

```
<tempalte>
  <div class="uk-block uk-block-primary uk-contrast page-header">
    <div class="uk-container-center">
      <h1 class="uk-heading-large">{{ header }}
      <small v-if="subHeader">{{ subHeader }}</small>
    </h1>
  </div>
</div>
</tempalte>
<script>
  export default {
    props: ['header', 'subHeader']
  }
</script>
```

6.3.1 搜索区的组件化

搜索区组件化的思路与上文一致，在此略过不表，先看看它的代码：

搜索区

```
<template>
```



```

<div class="uk-form">
  <div class="uk-form-icon">
    <i class="uk-icon-search"></i>
    <input type="search"
      class="search-box uk-form-width-large"
      :placeholder="placeholder"
      @keyup.enter="$emit('search', $event.target.value)"
      :value="terms" />
  </div>
</div>
</template>
<script>
export default {
  name: 'SearchBox',
  props: ['terms', 'placeholder']
}
</script>

```

这里有一点要注意，搜索区中的 `input` 与 `App.vue` 中的 `terms` 变量进行了双向绑定，当用户输入搜索关键字时就会自动更新 `terms`。在 `Vue2` 以前我们还可以使用 `.sync` 这个属性修饰符来使属性也能具有双向绑定功能，但在 `Vue2` 中这一功能完全被废除了，组件的状态是不可变的（Immutable），只能输入，不能在组件实例内的任何地方进行修改，一旦我们对 `props` 定义的变量进行修改，马上就会触发一个异常。

双向绑定已不能用于自定义组件的问题应当如何解决？答案是**事件**，组件内对变量做出修改只能向父容器发出一个事件，将修改值传递给父组件，由真正维护状态的组件来更新值。

所以在 `SearchBox` 中将原有的 `v-model="terms"` 换成了：

```

<input :value="terms"
  @keyup.enter="$emit('search', $event.target.value)" />

```

`:value` 用于将外部输入的属性值写到 `input` 内，当用户敲击键盘的回车键时用 `$emit` 方法发出一个 `search` 事件，通知父容器进行处理。

那 `App.vue` 内的代码就要进行这样的修改：

```

<search-box :terms="terms"
  placeholder="请输入您要筛选的书名"
  @search="terms=$event">
</search-box>

```

虽然这样写比原来的代码多了一些，但为了**状态共享**，这一点付出也是值得的。

6.3.2 母板组件

将这两个组件重构并封装成为新组件后，页面的总代码行数减少得并不多。此时，如果我们从上自下仔细地阅读一次代码，会发现最大量的代码都是一些 UIKit 布局结构代码，将这些代码以从属关系折叠起来，正好是我们一开始划分出来的几大功能区代码。

如果将这些功能区用插槽 slot 取代后会得到这样一种结构：

```
<div>
  <div slot="header">
    <!-- 页头 -->
  </div>

  <!-- 工具栏 -->
  <div slot="counting">
    <!-- 图书统计 -->
  </div>
  <div slot="search">
    <!-- 搜索框 -->
  </div>
  <div slot="buttons">
    <!-- 按钮组 -->
  </div>
  <!-- 工具栏 -->
  <!-- 正文（默认插槽） -->
    <!-- 图书数据表格 -->
    <!-- 对话框-->
    <!-- 图书编辑/新建 数据表单 -->
    <!-- 对话框-->
  <!-- 正文 -->
  <div slot="footer">
    <!-- 页脚-->
  </div>
</div>
```

从这个角度来看，整个页面其实就是一个更高层级的容器类页面——母板页。母板页负责封装 UIKit 定义的各种容器类布局，最终以插槽形式进行内容分发。根据上面的分析，完整的母板页的代码如下所示。

```
<template>
  <div class="page">
    <slot name="header">
      <page-header :header="title">
```

```

      :sub-header="subTitle">
    </page-header>
  </slot>
  <div class="content">
    <div class="uk-grid uk-margin-large-bottom">
      <div class="uk-width-3-4">
        <div class="uk-grid">
          <div class="uk-width-1-2">
            <slot name="counting"></slot>
          </div>
          <div class="uk-width-1-2">
            <slot name="search"></slot>
          </div>
        </div>
      </div>
      <div class="uk-width-1-4">
        <div class="uk-float-right">
          <slot name="buttons"></slot>
        </div>
      </div>
    </div>
    <slot></slot>
  </div>
  <div class="uk-margin-top uk-text-center">
    <slot name="footer"></slot>
  </div>
</div>
</template>
<script>
import PageHeader from './pageheader'
export default {
  name: 'ViewPage'
  props: ['title', 'subTitle'],
  components: {PageHeader}
}
</script>

```

这样一封装，我们就无须再去理会现在用的到底是 UIKit 还是 Bootstrap 了，要改变布局的样式、位置，可以在组件内不改变插槽名称情况下进行了，这样就能从很大的程度上将界面代码与 UIKit “解耦”了。

由此及彼，既然我们可以制作一个专门用于数据维护（CRUD）的母板页组件，那么还可以制作如登录、相册、博客、产品展台等各种具有较高重用性的母板组件，形成我们

的母板库，以便于在各个项目中使用。

母板组件实质上是借用了像 razor、jade、jinja 这一类服务端模板中的“母板”特性，因为 Vue 是一个面向组件的开发框架，用它特有的插槽（slot）将“布局”（Layout）封装成为一种组件，为大规模页面开发带来了非常大的便利性。从页面布局的角度我们称之为母板，但如果从局部入手又可以得到各种容器类组件，例如 Panel、Tabs、SideBar，等等。

6.3.3 重构模态对话框组件

在第4章组件化的设计与实现方法中我们实现了一个模态对话框组件，先来回顾一下它的代码：

```
<template>
  <div class="dialog-wrapper"
    :class="{ 'open': is_open }">
    <div class="overlay" @click="close"></div>
    <div class="dialog">
      <div class="header">
        <slot name="header"></slot>
      </div>
      <slot></slot>
      <slot name="footer"></slot>
    </div>
  </div>
</template>
<script>
  import "./dialog.less"
  export default {
    data () {
      return {
        is_open: false
      }
    },
    methods: {
      open () {
        if (!this.is_open) {
          // 触发模态窗口打开事件
          this.$emit('dialogOpen')
        }
        this.is_open = true
      },
      close () {
```



```

        if (this.is_open) {
          // 触发模态窗口关闭事件
          this.$emit('dialogClose')
        }
        this.is_open = false
      }
    }
  }
}
</script>

```

为了减少篇幅此处略去对话框的样式表部分。

UIkit 的模态对话框比我们上面这个纯手工打造的粗陋无比的对话框有更多的效果，例如一些动画效果、更美观的样式等。在不改变这个组件的用法接口的前提下，我们在组件内部将其改写为一个基于 UIkit 模态对话框的组件。

请谨记一点，改写组件一定要避免随意地改变组件的外部接口，因为这样做会让原有的代码出现不可预测的异常，这对于一个已被工程化的项目来说是危险的！

```

<template>
  <div class="uk-modal"
    ref="modal">
    <div class="uk-modal-dialog">
      <slot name="header">
        <div slot="header" class="uk-modal-header">
          <a class="uk-modal-close uk-close uk-float-right"></a>
          <h2 class="uk-display-inline">{{ headerText }}</h2>
        </div>
      </slot>
      <slot></slot>
      <slot name="footer"></slot>
    </div>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        dialog: undefined
      }
    },
    props: ['headerText'],
  }

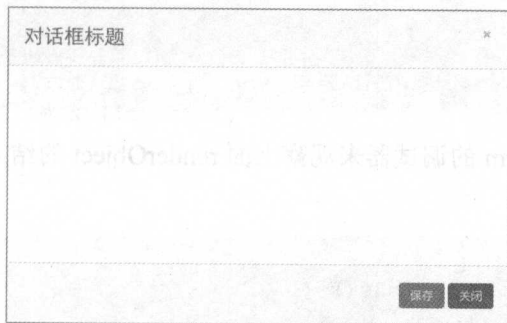
```



```
mounted () {  
  this.dialog = this.$ui.modal(this.$refs.modal)  
  var self = this  
  this.dialog.on('show.uk.modal', () => self.$emit('dialogOpen'))  
  this.dialog.on('hide.uk.modal', () => self.$emit('dialogClose'))  
},  
methods: {  
  open () {  
    this.dialog.show()  
  },  
  close () {  
    this.dialog.hide()  
  }  
}  
}
```

你会发现其实这样改写后代码变得更少了，也省去了自己写样式表的烦恼，而且改写的内容并不多，只是 `mounted` 钩子的内容变化了一下，另外为 `header` 插槽增加了默认的对话框头，子组件也可以声明这个插槽，对默认的头内容进行重定义。

以下就是改写模态对话框后的效果：



现在这种模态对话框的形式更适合用户的使用习惯，同时也可以在不同的页面内重用了。

6.3.4 高级组件与 Render 方法

本书开篇也提到过 Vue2 为了提高运行效能是集成了 VirtualDOM 的，之前我们所使用的 `template` 属性和 `<template>` 标记最终都会被编译为一个 `Render` 对象，这才是 Vue2 的真相！但官方网站上对 `Render` 的很多高级用法都讳莫如深，少之又少。能在百度或者谷歌上找到

的关于 `Render` 方法的案例更是不多。除非你曾经做过 `React` 的开发，否则一开始很难体验到 `Render` 方法带来的好处，只能从官网文章中了解到 `Render` 是一种用 JS 代码绘制组件的方法，用以取代 `<template>` 而已。

`Render` 方法需要在好几个组件的应用中进行深度的解读，如果将它的使用方法直接从官网贴到这里，对你是一点帮助的，这样的话本书就毫无价值可言了。

在开始通过真实示例讲述用 `Render` 方式开发 `Vue` 组件之前，我们需要对 `Render` 方法的基本原理与知识进行学习，这样才有助于我们理解 `Virtual DOM` 的方法论。

首先，`Vue2` 提供了一个很有趣的全局方法，叫作 `compile`（编译），这个方法就是将 `HTML` 模板的字符串编译为一个 `Render` 对象。

例如：

```
let renderObject = Vue.compile(`<div>
  <h1>模板</h1>
  <p v-if="message">
    {{ message }}
  </p>
  <p v-else>
    尚无消息
  </p>
</div>`)
```

我们可以用 `WebStorm` 的调试器来观察上面 `renderObject` 的结果，会发现它是一个拥有两个方法的对象：

```
{
  render: function anonymous() {
    with(this){
      return _h('div',[_m(0),(message)?_h('p',[_s(message)]):_h('p',["尚无消息"])]))
    }
  },
  staticRenderFns:[
    _m(0): function anonymous() {
      with(this){return _h('h1',["模板"])}
    }
  ]
}
```

如果想试试这个编译效果，可以访问 Vue 官网的一个在线小工具：<https://vuejs.org/v2/guide/render-function.html#Template-Compilation>。

在一般情况下，我们并不会在代码内直接执行 `compile` 来生成这个 `Render` 对象，这个操作是由 `Vue2` 的运行时帮我们完成的。由上述的内容我们可以了解到，`template` 模板是为了让我们可以像 `Angular` 那样来使用双向绑定、指令标记、过滤器等面向 `HTML` 标记的用法，而它们最终还是会变成 `Render` 函数。也就是说，`Vue2` 组件有另一种写法：纯 `JS` 代码。这种组件页可以看作一个代码型的单页组件，与 `*.vue` 编写的组件不同的是，使用 `Render` 方法的组件是一个标准的 `AMD` 模块。具体格式如下：

```
export default {
  // 组件名，这是必需的，如果没有的话会报出“渲染匿名组件”的异常
  // 另外这个组件名同时定义了页面中使用的标签
  // 例如这个 UkButton 组件使用时就是 <uk-button>
  name: 'UkButton',
  props: [], // 公共属性定义
  data: {    // 内部变量定义
    //...
  },
  methods: {}, // 其他的定义与 .vue 单页组件的定义是一致的

  // .. 略去

  render (createElement) { // 与 .vue 单页组件不同的只在于此
    return createElement('button', {
      class: {
        'uk-button': true
      }
    })
  }
}
```

这种纯 `JS` 方式的组件与普通的单页式组件 (`*.vue`) 唯一不同的地方是，普通的单页式组件需要 `<template>` 模板或者声明 `template` 属性，而纯 `JS` 方式的组件只需要定义 `Render` 方法。

createElement 函数

`Render` 方法会传入一个 `createElement` 函数，它是一个用于创建 `DOM` 元素或者用于实例化其他组件的构造方法。`Render` 方法必须返回一个 `createElement` 函数的调用结果，也就是模板内的顶层元素（这个方法在 `Vue2` 的习惯性使用中经常用 `h` 来命名）。

它有以下的用法：

1. 构造 DOM

```
export default {  
  // ...省略  
  render (createElement) {  
    const menu_items = [ "首页", "搜索", "分类", "系统" ]  
    return createElement('ul',  
      {  
        class: { 'uk-nav':true }  
      },  
      menu_items.map(item => createElement('li', item)))  
  }  
}
```

上述的 Render 方法用<template>来写的话应该如下所示。

```
<template>  
  <ul>  
    <li v-for="item in menu_items">  
      {{ item }}  
    </li>  
  </ul>  
</template>
```

2. 实例化 Vue 组件

```
import UkButton from 'components/button'  
  
export default {  
  // ... 省略  
  render (createElement) {  
    return createElement('div', [  
      createElement(UkButton, {  
        class: { 'confirm-button':true },  
        domProps : { innerHTML: "确定" },  
        propsData: { color: "primary", icon: "plus"}  
      })  
    ])  
  }  
}
```

对照为模板的语法，则为：

```
<template>  
  <div>
```

```

    <uk-button class="confirm-button"
      color="primary"
      icon="plus" >确定</uk-button>
  </div>
</template>

```

有了以上的对照，你是否已经清楚地知道为何 Vue2 一定需要有“顶层元素”的存在了（Vue1.x 是没有顶层元素限制的）？因为 `Render` 只能返回一个 `createElement`，这就是真相。

使用 `Render` 与 `template` 的区别有两点：首先在 `Render` 方法内是不能再使用任何指令标记的，指令标记从本质上说只是用 HTML 的方式表示某一种 JS 的语法功能，例如 `v-for` 就相当于 `map` 函数，`v-if` 就相当于条件表达式，两者的思维模式是完全不一样的。其次，对组件或网页元素的属性赋值是通过 `createElement` 函数的第二个参数 `data` 进行的，`domProps` 会向元素传递标准的 DOM 属性，而 `propsData` 则用于对其他的 Vue 组件的自定义属性（`props` 内的定义）进行赋值。

从这个角度来对照解释是不是感觉思路开阔了很多？用 `Render` 方法乍一看可能觉得所有指令标记都没有，之前 Vue 的好多基础性的技巧都不能用了，但实质上 `Render` 方法却能让我们更灵活地用 JS 代码去做更多的控制，避免了从 JS 实例到 HTML 属性这样的一种思维的转换。

接下来我们就全面地看看 `createElement` 的定义，让我们能对它有一个更深的理解：

```
createElement(tag, data, children)
```

返回值——`VNode`。

参数说明：

名 称	类 型	说 明
Tag	String/Object/Function	一个 HTML 标签，组件类型，或一个函数
Data	Object	一个对应属性的数据对象，用于向创建的节点对象设置属性值
Children	String/Array	子节点（ <code>Vnodes</code> ）

这三个参数中要对 `data` 参数进行附加说明，向构造的 `VNode` 对象设置文本时可以直接传入字符串，例如：

```
createElement('div', '这是行内文本')
```

那它的输出结果就是：

```
<div>这是行内文本</div>
```


当 data 对象是一个 Object 类型的话就相当复杂了，可以参考下表：

属 性	名 称	说 明
Class	Object	和 v-bind:class 一样格式的对象
Style	Object	和 v-syle:class 一样格式的对象
Attrs	Object	设置 HTML 属性对象
Props	Object	设置组件的属性对象
domProps	Object	设置 DOM 的内置属性
On	Object	设置组件事件处理方法如 click、change 等
navtiveOn	Object	仅对于组件，用于监听原生事件，而不是组件使用 vm.\$emit 触发的事件
directives	Object	自定义指令标记对象
Slot	String	如果子组件有定义 slot 的名称
Key	String	—
Ref	String	定义当前组件的引用名称

以下是 data 属性的使用范例：

```
// 假设有 EmptyHolder 自定义组件
createElement(EmptyHolder, {
  // 和 v-bind:class 一样的 API
  'class': {
    'uk-container': true,
    'uk-container-center': false
  },
  // 和 v-bind:style 一样的 API
  style: {
    color: 'red',
    fontSize: '14px'
  },
  // 正常的 HTML 特性
  attrs: {
    id: 'page-container'
  },
  // 组件 props
  props: {
    emptyText: '尚无任何内容'
  },
  // DOM 属性
  domProps: {
    innerHTML: '请手动刷新'
  },
}
```

```

// 事件监听器基于 "on"
// 所以不再支持如 v-on:keyup.enter 的修饰器
// 需要手动匹配 keyCode
on: {
  click: this.clickHandler
},
// 仅对于组件，用于监听原生事件，而不是组件使用 vm.$emit 触发的事件
nativeOn: {
  click: this.nativeClickHandler
},
// 自定义指令。注意事项：不能对绑定的旧值设置
// Vue 会持续追踪
directives: [
  {
    name: 'my-custom-directive',
    value: '2'
    expression: '1 + 1',
    arg: 'foo',
    modifiers: {
      bar: true
    }
  }
],
// 如果子组件有定义 slot 的名称
slot: 'name-of-slot'
// 其他特殊顶层属性
key: 'myKey',
ref: 'myRef'
})

```

此时你可能又会提出一个疑问，从代码量与代码的直观程度来讲，Render 方法显得很累赘，可读性又非常差。而且一旦输出的组件结构复杂，这个 Render 方法就会变得极为可怕。如果想象不到它的可怕程度达到哪一级别，在下文中讲述的一个 `datatable` 组件中用 `createElement` 函数来写一次，作为它可怕的证明：

```

export default {
  // ... 省略
  render (createElement) {
    let _fs = this.fields

    // 显示排序标记
    const sortFlag = header => createElement('span', {
      class: {
        'hidden': this.sortingKey !== header.name,

```

```
      'uk-icon-sort-asc': this.direction === 'asc',
      'ui-icon-sort-desc': this.direction === 'desc'
    }
  })

  // 绘制表头
  const colHeader = (header, index) => {
    var dataOpts = {
      class: {
        'uk-text-center': true,
        'disable-select': true,
        'sorting': this.sorted(header.name)
      },
      on: {
        click: () => this.sortBy(header.name)
      }
    }

    if (index === 0) {
      dataOpts = _.extend({}, dataOpts, {
        attrs: {
          colspan: 2
        }
      })
    }

    return createElement('th', dataOpts, [createElement('div', {
      domProps: {
        innerHTML: header.title
      }
    }, [sortFlag(header)])])
  }

  const toolCellNode = item => createElement('td', [createElement('input', {
    domProps: {
      type: 'checkbox'
    },
    attrs: {
      'data-id': item[this.keyField]
    },
    on: {
      change: e => this.selectionChanged(item, e)
    }
  })])
}
```

```

// 绘制单元格
const cellNodes = item => [toolCellNode(item)].concat(this.dataFields.
map(df => {
  if (_fs[df.name]) {
    // 动态装配组件
    return createElement(_.extend({},
      _fs[df.name].data.inlineTemplate, {
        name: 'CustomField',
        props: ['name', 'item']
      }), {props: {name: df.name, item: item}})
  }
  else {
    return createElement('td', {}, [createElement('div', {
      class: {
        'fill': true,
        'sorting': this.sorted(df.name)
      },
      domProps: {
        innerHTML: item[df.name]
      }
    })])
  }
}))

// 绘制行对象
const rowNodes = this.dataItems.map(item => createElement('tr', {},
cellNodes(item)))

return createElement('table', {
  class: {
    'uk-table': true,
    'uk-table-striped': true
  }
}, [
  createElement('thead', [createElement('tr', {}, this.dataFields.
map(colHeader))]),
  createElement('tbody', [rowNodes])
])
}
}

```

上述的代码旨在展示如果纯粹用 `createElement` 来构造组件的话代码将会有多么痛苦！

JSX 语法

使用 `createElement` 编写组件显得非常 `HardCode`，为了能弥补这一缺陷又能享受 `Render` 所带来的可编程性，我们可以用 `React` 特有的 `JSX` 文件的语法。`JSX` 在 `Vue` 中的出现几乎就是为了给 `React` 开发者带来福音，同时也是简化 `createElement` 方法的最好途径。

我们用 `JSX` 语法来写一个简单的导航栏，并以此为例讲述 `JSX`：

```
export default {
  props: {
    items: {
      type: Array,
      default () => [] // 当属性默认值为对象时要用函数返回
    }
  },
  render (h) {
    return <ul class={
      'uk-nav':true
    }>
      {
        this.items.map(item => <li>
          <a href={ item.url }>
            { item }
          </a>
        </li>)
      }
    </ul>
  }
}
```

上述代码对于没有学过 `JSX` 的开发者来说可能有点怪异，简单理解 `JSX` 就是一种用 `HTML` 模板代码取代 `createElement` 函数的一种语法，模板中的变量通过 `{}` 引用实例内的变量，这有点像字面量引用。还有一点就是元素属性可以像 `createElement` 那样进行赋值绑定，但语法上有点变动：

```
render (h) {
  return (
    <div
      // 常规的 HTML 属性
      id="foo"

      // DOM 属性全部需要增加 domProps- 前缀
      domProps-innerHTML="bar"
```



```

// 事件绑定则需要在事件之前加入 on- 或者 nativeOn- 前缀
on-click={this.clickHandler}
// 事件绑定不能使用表达式，只能用方法的引用
nativeOn-click={this.nativeClickHandler}

// 样式类绑定一定要用一个对象实例，true 表示添加样式类，false 表示删除样式类
class={{ foo: true, bar: false }}
style={{ color: 'red', fontSize: '14px' }}
key="key"

// 给当前组件进行引用命名，可以在代码的 this.$refs 中引用 DOM 实例
ref="ref"

// 指定父组件的插槽名称
slot="slot">
</div>
)
}

```

请注意：JSX 中所有在 template 中的语法、指令、过滤器都将失效，而且它还不是像 JSX 那样有 #if 等关键字，条件判断等语法都需要自己通过函数来实现。

安装 JSX 支持

Vue 中的 JSX 并不是 Vue 原生自带的，它必须安装一个由 Vue 官方提供的 babel-plugin-transform-vue-jsx (<https://github.com/vuejs/babel-plugin-transform-vue-jsx>) 库才能被 Babel 编译器识别它。babel-plugin-transform-vue-jsx 具体的安装方法如下：

```

$ npm install\
  babel-plugin-syntax-jsx\
  babel-plugin-transform-vue-jsx\
  babel-helper-vue-jsx-merge-props\
  --save-dev

```

安装完成后还需要将 .babelrc 文件的内容改成以下方式：

```

{
  "presets": ["es2015"],
  "plugins": ["transform-vue-jsx"]
}

```

做完这两步 JSX 就能使用了。

这就是属于 Vue 的 JSX！虽然与 React 相比会觉得有很多地方都被简化过了，回想一下 Vue 的功能路径，不就是因为它简化了 Angular 吗？撇开 Angular 和 React 不进行对比的话，

这也是给我们提供了另一种编写组件的直接途径，作为一个技术使用者，保持“海纳百川，有容乃大”的心态才是王道。

另外，目前 Vue 版 JSX 的最大痛点是它没有办法调试，因为它没有办法设置断点，有些情况下断点的定位是完全不准确的，估计是 babel-plugin-transform-vue-jsx 生成的 source-map 有缺陷导致的。

小结

Vue 官方仍然推荐我们使用 `<template>` 方式来制作组件，但 Render 方法与 JSX 的引入却给予了我们更大的灵活性，对于一些复杂性很强的组件，Render 方法是一种有效的简化手段。在一开始使用它的时候可能有一点不习惯，但由于它本身就是 Vue2 的核心基础，熟悉以后反而会觉得代码更简洁了，会更偏向于使用它。本节只是对 Render 方法与 JSX 进行背景知识的学习，所以没有过多的示例讲解，而在本书后面的章节中将会有更多的组件采用 Render 方法进行开发，在其中会继续将一些官方没有透露的使用技巧进行揭秘。

6.3.5 UIkit 按钮

在这一小节中我们就使用 Render 方法来编写一个常用的按钮组件。现在 App.vue 的页面内有好几个这样的按钮代码：

```
<button class="uk-button uk-button-danger"
  @click.prevent="事件处理方法">
  <i class="uk-icon-trash"></i> 按钮标题
</button>
```

这种写法的坏处是我们必须要记住 UIkit 的样式名，而且按钮又是交互界面的最基础元素，要使用的地方很多，所以很有必要将这种写法进行简化，在没有组件化前先来写出这个按钮组件的使用方法：

```
<uk-button icon="trash"
  color="danger"
  @click="事件处理方法">按钮文字</uk-button>
```

换成以上的用法是不是会感觉清爽很多？那我们就按这个用法来写按钮组件，在 components 目录下建立一个 button.js 文件，代码如下：

```
export default {
```

```
name: 'UkButton',
props: {
  active: {
    type: Boolean,
    default: false
  },
  disabled: {
    type: Boolean,
    default: false
  },
  color: {
    type: String,
    default: ''
  },
  icon: {
    type: String
  },
  rightIcon: {
    type: String
  },
  size: {
    type: String,
    default: ''
  },
  width: {
    type: String,
    default: ''
  }
},
render (h) {
  const data = {
    class: {
      'uk-button': true,
      'uk-active': this.active,
      ['uk-width-${this.width}']: this.width,
      'uk-button-primary': this.color === 'primary',
      'uk-button-success': this.color === 'success',
      'uk-button-danger': this.color === 'danger',
      'uk-button-link': this.color === 'link',
      'uk-button-mini': this.size === 'mini',
      'uk-button-small': this.size === 'small',
      'uk-button-large': this.size === 'large'
    }
  }
}
```

```

    let clickHandler = (e) => {
      e.preventDefault()
      this.$emit('click')
    }

    return (
      <button disabled={ this.disabled }
        on-click={ clickHandler }
        {...data}>
        <uk-icon name={this.icon}></uk-icon> { this.$slots.default }
        {this.rightIcon && <uk-icon name={this.rightIcon}></uk-icon>}
      </button>
    )
  }
}

```

我为这个按钮组件加入了大量的属性，这样做的目的是为了避免使用这个按钮组件时去记忆 UIKit 提供的那些专用的 CSS 类，从而大量地减少代码的重复性以达到降低代码量的目的。

通过 Vue 组件的属性设定，我们令按钮组件支持了属性的绑定，使得外部调用 UkButton 的组件可以通过变量来控制按钮的样式。

最后，在渲染时我使用了一点函数式的编程技巧，因为 JSX 中是没有 if 标记的，如果要进行条件判断就会显得很麻烦，所以我采用条件表达式来熔断 if 语句，这可以说是函数式编程中必学的技巧了。

上文中：

```
this.rightIcon && <uk-icon name={this.rightIcon}></uk-icon>
```

可以理解为这样的伪代码：

```

if this.rightIcon
  return <uk-icon name={this.rightIcon}></uk-icon>

```

用 TDD 为 Vue 的组件化开发保驾护航

按钮组件是一个极为常用的组件，所以我们不要将它放到某一个界面上时才来调试它的用法，而是应该编写一个组件测试，先通过组件的单元测试后才将其大量地运用到我们的界面中。

首先，在 `~/src/test/unit/spects` 中创建一个 `UkButton.spec.js` 的按钮组件测试文件，具体的内容如下所示。


```

import UkButton from 'components/button'
import {getVM} from '../helpers'
import _ from 'lodash'

describe('UkButton', () => {
  it('$mount', () => {

    const clickHandler= sinon.spy()

    let vm = getVM(h => <uk-button icon="disk"
      right-icon="edit"
      active={true}
      size="large"
      on-click={ clickHandler }>
        保存
      </uk-button>
    )

    const cls=vm.$el.getAttribute('class')

    expect(cls).to.include('uk-button-large')
    expect(cls).to.include('uk-active')

    expect(vm.$el.querySelector('.uk-icon-edit')).to.exist
    expect(vm.$el.querySelector('.uk-icon-disk')).to.exist

    expect(_.trim(vm.$el.textContent)).to.eqls('保存')

    window.$(vm.$el).trigger('click')
    expect(clickHandler).to.have.been.called

  })
})

```

不少程序员都知道测试很重要，但是真正开始写测试时却无从入手，不知道应该测试什么。其实单元测试的原则非常简单，只需要理解“幂等性”，所谓的“幂等性”就是“输出决定于输入”，这个概念源自于函数式编程，但我认为将它应用于组件的单元测试同样适用。

界面组件必然是“幂等”的，因为 Vue 组件的输出结果必然受到输入属性控制，换一种说法就是一个正确的组件其输出结果必然与我们期待它输出的效果是一致的。

由此，我们就可以得到一个非常简单的组件测试思路：检测我们所期望组件最终渲染

的结果是否与预期一致。

以下的代码就是 `UkButton` 组件的输入，同时它也是在页面上真正被使用时的效果：

```
const clickHandler= sinon.spy()

let vm = getVM(h => <uk-button icon="disk"
  right-icon="edit"
  active={true}
  size="large"
  on-click={ clickHandler }>
  保存
</uk-button>
)
```

然后判断它是否将 `UIKit` 的样式类添加到 `button` 元素上：

```
expect(cls).to.include('uk-button-large')
expect(cls).to.include('uk-active')
```

检测是否输出左右两个图标元素：

```
expect(vm.$el.querySelector('.uk-icon-edit')).to.exist
expect(vm.$el.querySelector('.uk-icon-disk')).to.exist
```

最后通过 `jQuery` 触发元素上的点击事件，模拟用户点击按钮，检测按钮是否能成功发出点击事件：

```
window.$(vm.$el).trigger('click')
expect(clickHandler).to.have.been.called
```

事实上，并不是先开发出组件再来写测试的。为了读者能更容易在阅读过程中理解这个思路才特意将测试放在实现之后。在真正的开发中应该先写出上述的这个测试，在没有实现这个组件之前就能先考虑清楚组件需要输入什么，输出什么，触发什么事件，通过测试来进行设计才是 `TDD` 的意义所在。

6.3.6 通用表格组件

对其他简单组件完成封装之后，剩余具有最多代码量的地方就是表格 `table` 了。仔细分析数据表格的代码就不难看出，所有的代码都是为了提供行与列的数据显示与操作。以下是对表格组件的分析设计思路。

表格列处理的分析思路

(1) 对列信息进行枚举并渲染列头元素——从原来的表格代码中将列的信息提取为一个具有 name (名称) 和 title (标题) 的数组, 并将此数组作为外部参数输入到表格的自定义属性 data-fields, 这样就可以在表格组件内用一个 v-for 的枚举循环渲染所有的列头元素。

(2) 按列排序, 并显示列的排序状态——由此可得出点击列头时应发出排序事件 selectionChanged。

表格行处理的分析思路

(1) 分别对行与列进行枚举并渲染字段值的相关元素——将数据行传入到表格组件的自定义属性 data-items, 由表格组件对所有的行进行统一的渲染处理。

(2) 提供多行选定——当行的选中状态发生改变时可发出事件 sort。

(3) 要提供一种可定制化的手段, 能从组件的外部重定义字段渲染模板。

首先, 根据以上的分析结果来设计表格组件的使用接口:

```
<data-table :data-items="books"
  :data-fields="[
    {name: 'name', title: '书名'},
    {name: 'category', title: '分类'},
    {name: 'published', title: '发布日期'}
  ]"
  @selection-changed="selectionHandler"
  @sort="sortHandler">
</data-table>
```

然后, 建立一个表格组件的代码框文件:

```
// components/datatable.js
export default {
  name: 'DataTable',
  render (createElement) {
  }
}
```

接着, 为 DataTable 组件编写一个单元测试, 并将上文中组件的方式写入到单元测试中:

```
// test/unit/spects/datatable.spec.js
import DataTable from 'components/datatable'
import Vue from 'vue'
import BooksData from 'src/fixtures/items.json'
```

```

import compileComponent as c from '../helper'

describe('datatable', () => {
  it('应该自动根据输入数据行与列定义输出正确的表格结构', () => {
    let vm = c(`<div>
      <data-table :data-items="items"
        :data-fields="fields">
    </data-table></div>`,
    { DataTable },
    {
      items: BookData,
      fields: [
        {name: 'name', title: '书名'},
        {name: 'category', title: '分类'},
        {name: 'published', title: '发布日期'}
      ]
    })

    expect(vm.$el.querySelector('tbody>tr').length).toEqual(BooksData.length)
    expect(vm.$el.querySelector('thead>tr>th').length).toEqual(3)
  })
})

```

按照测试中的定义先实现表格组件的属性代码：

```

export default {
  name: 'DataTable',
  props: {
    dataItems: {
      type: Array,
      default: []
    },
    dataFields: {
      type: Array,
      default: []
    }
  }
}

```

我们可以从原来的代码中将排序与选择部分的方法封装到 `DataTable` 中：

```

import _ from 'lodash'

export default {
  name: 'DataTable',

```

```

props: {
  // ... 省略
},
data () {
  return {
    sortingKey: '',
    direction: ''
  }
},
methods: {
  sorted (key) {
    return key === this.sortingKey
  },
  sortBy (key) {
    if (key === this.sortingKey) {
      this.direction = this.direction === 'asc' ? 'desc' : 'asc'
    }
    this.sortingKey = key
    this.$emit('sort', {
      field: key,
      dir: this.direction
    })
  },
  selectChanged (item, e) {
    item.selected = e.target.checked
    if (e.target.checked) {
      this.selection.push(item[this.keyField])
      this.selection = _.uniq(this.selection)
    } else {
      this.selection = _.reject(this.selection, b => item[this.keyField]
=== b)
    }

    this.$emit('selection-changed', {
      selection: this.selection,
      item: item
    })
  }
},
}

```

接下来我们就要实现表格组件的渲染部分了，表格组件的渲染逻辑比较复杂，这一点在 6.3.4 节高级组件与 Render 方法中曾用 `createElement` 方法展示过一次，而现在采用 `Render` 方法和 `JSX` 来重新编写一次，在阅读完本节后建议读者可以将本节的代码与 6.3.4 节用

createElement 方法实现的 DataTable 进行对照，这样有益于更好地理解 JSX。

以下代码是对表头的渲染逻辑：

```
render (h) {
  return <table class={{
    'uk-table': true,
    'uk-table-striped': true
  }}>
    <thead>
      <tr>
        {
          // 循环输出表格的列头元素
          this.dataFields.map((header, index) => (
            <th class={{
              'uk-text-center': true,
              'disable-select': true,
              'sorting': this.sorted(header.name)
            }}
              colSpan={index === 0 ? 2 : 1}
              on-click={ (header) => this.sortBy(header.name) }>
                <div>
                  { header.title }
                  <span class={{
                    'hidden': this.sortingKey !== header.name,
                    'uk-icon-sort-asc': this.direction === 'asc',
                    'ui-icon-sort-desc': this.direction === 'desc'
                  }}>
                </span>
                </div>
              </th>
            )
          )
        }
      </tr>
    </thead>
    <tbody>
      {
        this.dataItems.map(item => (
          <tr>
            <td>
              <input type="checkbox"
                data-id={item[this.keyField]}
                on-change={ ($event) => this.selectionChanged(item,
$event) } />

```



```

        </td>
        {
          // 按照表格列结构统一输出字段值
          this.dataFields.map(df => {
            <td>
              <div class={{
                'fill': true,
                'sorting': this.sorted(df.name)
              }} on-click={ () => this.$emit('cell-click', propsData) }>
                { item[df.name] }
              </div>
            </td>
          ))
        }
      </tr>
    ))
  }
</tbody>
</table>
}

```

运行测试通过后，上文中分析的数据表格的基本功能已经完成。接下来需要为表格加入可独立定义的字段模板，使自定义列可以输出除了字段字面量外增加的其他更丰富的元素。

我们希望 **DataTable** 能以以下的方式来定义自定义输出字段：

```

<data-table :data-items="items"
  :data-fields="fields">
  <field name="name">
    <div>
      <!--自定义元素-->
    </div>
  </field>
</data-table>

```

Vue 提供了一个模板属性叫作 **inline-template**，向组件加入该属性后原组件的模板或 **Render** 方法将会失效并被 **inline-template** 指定的模板所取代，我们可以利用这一特性来制作一个组件外的自定义模板。先为这个自定义模板的用法编写单元测试：

```

it('应该正确输出自定义列并触发排序和行选事件', () => {
  let sortHandler = sinon.spy()
  let selectionChangeHandler = sinon.spy()
  let linkHandler = sinon.spy()

```

```

let vm = compileComponent(`

<data-table :data-items="items"
    :data-fields="fields"
    @sort = "sortHandler"
    @selection-change="selectionChangeHandler"
    @cell-click="linkHandler">
    <field name="name" inline-template>
      <div>
        <a>
          {{ item.name }}
        </a>
        <p>
          {{ item.isbn }}
        </p>
      </div>
    </field>
  </data-table></div>`, {
  methods: {
    linkHandler,
    sortHandler,
    selectionChangeHandler
  }
})

expect(vm.$el.querySelector('tbody>tr').length).to.eql(BooksData.length)
expect(vm.$el.querySelector('thead>tr>th').length).to.eql(3)
expect(vm.$el.querySelector('a').length).to.eql(BooksData.length)

$(vm.$el.querySelector('.custom-cell')[0]).trigger('click')
$(vm.$el.querySelector('thead>tr>th')[1]).trigger('click')

$(vm.$el.querySelector('tbody>tr:first>td>input')).trigger('click')

expect(sortHandler).to.have.been.called
expect(linkHandler).to.have.been.called
expect(selectionChangeHandler).to.have.been.called
})


```

上述代码中的 `field` 组件从用法上来说只是一个模板的占位符，它本身并不会进行渲染，我们只是为了通过它获知表格内是否具有自定义的模板，如果有就用该自定义模板来渲染数据行。

那么就要先定义一个 `field` 组件来“欺骗”Vue 以达到占位的效果：

```
// components/datatable.js
export default {
  components: {
    Field: {
      name: 'Field',
      props: ['name', 'item'],
      render: h => h('span')
    }
  }
}
```

当元素尚未被加载到 VirtualDOM 之前也就是在 `beforeMount` 钩子函数内，我们可以从 `this.$slots.default`（`DataTable` 的默认模板）中判断是否具有 `field` 名称的元素，如果有则从中获取自定义元素上的属性：

```
// components/datatable.js
export default {
  components: {
    Field: {
      name: 'Field',
      props: ['name', 'item'],
      render: h => h('span')
    }
  },
  // ... 省略
  data () {
    return {
      fields: {},
      // ... 省略
    }
  },
  beforeMount () {
    if (this.$slots.default) {
      let _fields = this.$slots.default.filter(node => node.tag === 'field')
      let self = this
      _fields.forEach(f => {
        self.fields[f.data.attrs.name] = f
      })
    }
  }
}
```

最后，重构 `Render` 方法，使用自定义模板来对行数据进行渲染：

```
render (h) {
```

```

return <table class={{
  'uk-table': true,
  'uk-table-striped': true
}}>
  <thead>
    <!--省略 输出表头-->
  </thead>
  <tbody>
    {
      this.dataItems.map(item => (
        <tr>
          <td>
            <input type="checkbox"
              data-id={item[this.keyField]}
              on-change={($event) => this.selectionChanged(item,
$event)}}/>
          </td>
          {
            this.dataFields.map(df => (
              <td>

                <div class={{
                  'fill': true,
                  'sorting': this.sorted(df.name)
                }} on-click={ () => this.$emit('cell-click', propsData) }>
                  {
                    this.fields[df.name]
                  }
                h(Vue.extend(_extend(this.fields[df.name].data.inlineTemplate, {
                  name: 'CustomField',
                  props: ['name', 'item']
                })), {
                  props: {
                    name: df.name,
                    item: item
                  }
                }) : item[df.name]
              )
            </div>
          </td>
        )
      )
    }
  </tr>
)}
}

```



```

    </tbody>
  </table>
}

```

这次重构将 `createElement` 方法与 `JSX` 混合使用，上述代码中的 `h` 实质上就是 `createElement` 方法。`h` 方法的第一个参数就是将 `field` 中定义的数据取出来（包括自定义模板 `inlineTemplate`），用 `Vue.extend` 创建一个新的 `Vue` 组件实例，将数据行 `item` 用代码方式绑定到 `props` 属性上传入：

```

h(Vue.extend(_.extend(this.fields[df.name].data.inlineTemplate, {
  name: 'CustomField',
  props: ['name', 'item']
})), {
  props: {
    name: df.name,
    item: item
  }
})

```

所以前文中提及 `field` 只是一个占位符的原因就在于此，它只用于接收用户定义的属性与模板的元素。在内部还是由 `DataTable` 的 `Render` 方法进行最终的渲染。

由于这个 `DataTable` 的实现思路比较复杂，为了方便地阅读这个组件的代码，以下是该组件的全部实现代码：

```

import _ from 'lodash'
import Vue from 'vue'
export default {
  name: 'DataTable',
  props: {
    dataItems: {
      type: Array,
      default: []
    },
    dataFields: {
      type: Array,
      default: []
    },
    keyField: {
      type: String,
      default: 'id'
    },
    autoGenerate: {
      type: Boolean,

```



```

    default: false
  },
  data () {
    return {
      sortingKey: '',
      direction: '',
      fields: {}
    }
  },
  beforeMount () {
    if (this.$slots.default) {
      let _fields = this.$slots.default.filter(node => node.tag === 'field')
      let self = this
      _fields.forEach(f => {
        self.fields[f.data.attrs.name] = f
      })
    }
  },
  methods: {
    sorted (key) {
      return key === this.sortingKey
    },
    sortBy (key) {
      if (key === this.sortingKey) {
        this.direction = this.direction === 'asc' ? 'desc' : 'asc'
      }
      this.sortingKey = key
      this.$emit('sort', {
        field: key,
        dir: this.direction
      })
    },
    selectChanged (item, e) {
      item.selected = e.target.checked
      if (e.target.checked) {
        this.selection.push(item[this.keyField])
        this.selection = _.uniq(this.selection)
      } else {
        this.selection = _.reject(this.selection, b => item[this.keyField]
        === b)
      }

      this.$emit('selection-changed', {

```

```

        selection: this.selection,
        item: item
      })
    }
  },
  render (h) {
    return <table class={{
      'uk-table': true,
      'uk-table-striped': true
    }}>
      <thead>
        <tr>
          {
            this.dataFields.map((header, index) => (
              <th class={{
                'uk-text-center': true,
                'disable-select': true,
                'sorting': this.sorted(header.name)
              }}
                colSpan={index === 0 ? 2 : 1}
                on-click={ (header) => this.sortBy(header.name) }>
                <div>
                  { header.title }
                  <span class={{
                    'hidden': this.sortingKey !== header.name,
                    'uk-icon-sort-asc': this.direction === 'asc',
                    'ui-icon-sort-desc': this.direction === 'desc'
                  }}>
                </span>
                </div>
              </th>
            )
          )
        </tr>
      </thead>
      <tbody>
        {
          this.dataItems.map(item => (
            <tr>
              <td>
                <input type="checkbox"
                  data-id={item[this.keyField]}
                  on-change={($event) => this.selectionChanged(item,

```

```

$event) } />
      </td>
    {
      this.dataFields.map(df => (
        <td>

          <div class={{
            'fill': true,
            'sorting': this.sorted(df.name)
          }} on-click={ () => this.$emit('cell-click', propsData) }>
            {
              this.fields[df.name]
            }
            h(Vue.extend(_.extend(this.fields[df.name].data.inlineTemplate, {
              name: 'CustomField',
              props: ['name', 'item']
            })), {
              props: {
                name: df.name,
                item: item
              }
            }) : item[df.name]
          </div>
        </td>
      ))
    }
  </tr>
))
}
</tbody>
</table>

},
components: {
  Field: {
    name: 'Field',
    props: ['name', 'item'],
    render: h => h('span')
  }
}
}

```

通过测试后，回到 `App.vue` 文件，按照测试中 `DataTable` 定义的用法删除原来所有的表格视图的代码，用 `DataTable` 组件加以取代，那么第一阶段的代码重构与组件提取就宣告完

成了。

6.4 表单的设计与实现

本节将直接针对单个数据对象的增加与编辑功能讲述具有更强交互性的数据表单。

接下来要实现的就是在 App.vue 中一直悬空的<!-- 图书编辑/新建 数据表单 -->的功能区块，相信到此你已经了解了 Vue 组件化方面的知识，我们已经不需要在 App.vue 上编写这个功能区域的内容，而是直接将表单看作一个复合型组件来设计与实现。

首先，设计这个表单组件的用法接口：

```
describe('book-form', () => {
  it('#save', () => {
    let formSaveHandler = sinon.spy()
    let book = {}
    let vm = getVM(<book-form book={book}
      @save={formSaveHandler}>
      </book-form>,
      {BookForm} )
  })
})
```

按照图书的数据结构定义编写组件模板：

```
<template>
  <div>
    <form class="uk-form uk-form-horizontal"
      v-if="current">
      <div class="uk-container uk-container-center">
        <ul class="uk-tab" data-uk-tab="{active:0,connect:'#tabContents'}">
          <li><a href="">通用</a></li>
          <li><a href="">摘要</a></li>
        </ul>
        <ul id="tabContents">
          <li>
            <div class="uk-form-row">
              <label class="uk-form-label"
                for="name">书名</label>
              <div class="uk-form-controls">
                <input id="name"
                  name="name">
```



```
        class="uk-form-width-large"
        autofocus="autofocus"
        v-model="current.name"/>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="isbn">书号</label>
    <div class="uk-form-controls">
        <input id="isbn"
            name="isbn"
            class="uk-form-width-large"
            v-model="current.isbn"/>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="price">售价</label>
    <div class="uk-form-controls">
        <input id="price"
            name="price"
            class="uk-form-width-large"
            type="number"
            v-model="current.price"/>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="catgeory">类别</label>
    <div class="uk-form-controls">
        <input id="catgeory"
            name="catgeory"
            class="uk-form-width-large"
            v-model="current.catgeory"/>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="published">出版日期</label>
    <div class="uk-form-controls">
        <input id="published"
            type="date"
            class="uk-form-width-large"
            name="published">
```



```

        v-model="current.published"
        placeholder="YYYY-MM-DD"/>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label">&nbsp;</label>
    <div class="uk-form-controls">
        <label>
            <input type="checkbox"
                v-model="current.is_published"/> 上市销售
        </label>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="pages">页数</label>
    <div class="uk-form-controls">
        <input id="pages"
            name="pages"
            class="uk-form-width-large"
            type="number"
            step="any"
            v-model="current.pages"/>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="authors">作者</label>
    <div class="uk-form-controls">
        <input id="authors"
            name="authors"
            class="uk-form-width-large"
            v-model="current.authors"/>
    </div>
</div>
</li>
<li>
    <!-- 摘要页的内容 -->
</li>
</ul>
</div>
</form>
</div>
</template>

```

```
<script>
  export default {
    props: ['book'],
    data () {
      return {
        current: {}
      }
    }
  }
</script>
```

6.4.1 计算属性的双向绑定

在数据结构中有一个作者（authors）字段属性，它是一个数组类型，那是否能用双向绑定将数组类型直接绑定到上呢？直接绑定肯定是不行的，因为的value属性只能接受字符串类型的值，此时可以运用计算属性（computed）在双向绑定的过程中进行类型的转换与匹配。

简单来讲就是在 authors 绑定到 input.value 之前将数组转化为一个以逗号分隔的字符串，反之则将字符串重新分解为数组存回 model 之中。

```
export default {
  data () {
    return {
      current : undefined
    }
  },
  computed: {
    authors: {
      get () {
        return this.current.authors ? this.current.authors.join(',') : ''
      },
      set (val) {
        this.current.authors = val.split(',')
      }
    }
  }
}
```

同理，在我们的数据中有一个 status 的字符串类型的属性字段，而且这个字段只有两种值：

(1) “上市销售”。

(2) 空字符串。

很明显，从数据逻辑上看这不应该是一个字符类型而应该是布尔类型，当这个值为布尔类型时我们就能用一个 `checkbox` 来作为这个字段值的输入控件，此时我们就需要对这个原有的字段进行转化，读取时获取布尔类型，而写入布尔值时则要向字段设置字符串，具体写法如下：

```
export default {
  data () {
    return {
      current : undefined
    }
  },
  computed: {
    authors: {
      // ... 省略
    },
    is_published: {
      get () {
        return this.current.status === '上市销售'
      },
      set (val) {
        this.current.status = val ? '上市销售' : ''
      }
    }
  }
}
```

在 Vue 的计算属性中，所谓的双向绑定从本质上与一个传统类属性的 `getter` 和 `setter` 无异。获取数据时可以通过 `getter` 进行一些附加处理，同理 `setter` 也发挥同样的作用。可见计算属性是 Vue 动态处理数据或状态的一种十分灵活多变的技巧与手法。

6.4.2 富文本编辑器组件的实现

富文本编辑器可以说是表单类应用场景中最不可或缺的组件之一。在本示例中“图书简介”字段就需要一个富文本编辑器来支持数据输入。

UIKit 最具吸引力的地方也就在此，它除了提供比 `Bootstrap` 更精美的样式、更简单的网页结构，还内置了很多在网页开发中经常用到的 JS 组件，`HtmlEditor` 便是其中之一。我

们只要将 `HtmlEditor` 这个 JS 组件用 `Vue` 进行组件化包装后就可以广泛地应用在我们的 `Vue` 项目中了。

设计 `HtmlEditor` 的用法

首先在 `~/test/unit/specs` 目录下为 `HtmlEditor` 创建单元测试文件 `htmleditor.spec.js`。接下来就是在测试文件中设计 `HtmlEditor` 组件的用法接口。`HtmlEditor` 应该如同其他的 `input` 类组件一样，通过 `value` 属性设置值，然后通过事件获取变更值。

消失了的修饰器：`sync` 和 `once`。`Vue 2.0` 的组件只能单向赋值，以往可以通过两路绑定的做法已经失效，所以只能通过事件将组件内部的数据变化暴露给父组件，因此以下的两种写法在 `Vue2` 中都将无效。

```
<html-editor :value.sync='modal.textValue'></html-editor>
<!-- 或者 -->
<html-editor :value.once='modal.textValue'></html-editor>
```

```
import UIKit from 'vue-uikit'
import Vue from 'vue'
import {getVM} from '../helpers'
import HtmlEditor from 'components/htmleditor'
import 'chance'

Vue.use(UIKit)

describe('htmleditor', () => {
  const valueChangedHandler = sinon.spy()
  const originalContent = Chance().paragraph()
  const editingContent = Chance().paragraph()

  let vm = getVM(h => (<html-editor value={originalContent}
                                on-change={valueChangedHandler}>

    </html-editor>), {HtmlEditor})

  let editor = vm.$children[0]
  // 取得 HtmlEditor 生成的 textarea 元素
  let textarea = editor.$el.querySelector('textarea')

  expect(textarea.textContent).to.eqls(originalContent)

  editor.value = editingContent

  $(textarea).trigger('input')
```



```
expect(valueChangedHandler).to.have.been.called
}))
```

在这个测试用例中，我用了 `chance` 库来产生测试数据，这个库非常有用，可以借助它产生随机数据来模仿实际的应用。在很多的测试用例中都会使用到它，为了可以简化测试用例，避免每次都引用，我们可以将 `chance` 这个库直接在 `webpack` 测试环境中配置使用 `ProviderPlugin` 生成全局的上下文。打开 `~/test/unit/karma.conf.js`，在 `webpack` 的 `plugins` 配置内加入 `Chance`：

```
plugins: [
  new webpack.ProvidePlugin({
    Chance : "chance",
    $: "jquery",
    jQuery: "jquery",
    "window.jQuery": "jquery",
    "window.$": "jquery"
  })
]
```

这样我们就不必每次都使用 `import 'chance'` 了。

请留意上述代码中获取 `HtmlEditor` 的 `Vue` 实例的方法：

```
let editor = vm.$children[0]
```

`vm.$children` 内存放的对象就是 `Vue` 的根对象下所有的 `Vue` 组件的实例，此处我们只有一个子实例，所以直接获取第一个元素。接下来就能像普通 `JS` 对象一样使用 `HtmlEditor` 实例了。最后加入一个 `spy`，检测当 `HtmlEditor` 的内容发生改变时是否会触发 `onchange` 事件，这样 `HtmlEditor` 的测试用例就编写完成了，接下来就是实现 `HtmlEditor` 组件了。

引入 `HtmlEditor` 的依赖

如果没有对 `Vue` 进行组件化，在网页上直接使用这个 `HtmlEditor` 也是一件非常麻烦的事，因为它有很多的依赖引用，一旦被我们组件化后就一了百了，以后都不用去记忆这些让人恼火的文件引用了。在 `HtmlEditor.vue` 组件代码内直接贴上以下的代码：

```
import "codemirror"
import "codemirror/mode/markdown/markdown"
import "codemirror/addon/mode/overlay"
import "codemirror/mode/xml/xml"
import "codemirror/mode/gfm/gfm"
import "marked"
import 'codemirror/lib/codemirror.css'
```



```
import "uikit/dist/js/components/htmleditor"
import "uikit/dist/css/components/htmleditor.css"
import 'uikit/dist/css/components/htmleditor.gradient.css'
```

CodeMirror 是 HtmlEditor 使用的最主要的依赖包，直接 import 后还得在 webpack 的配置中增加 CodeMirror 对 window 全局变量的映射。以下是修改 webpack 的配置以支持 CodeMirror 的具体方法：

```
plugins: [
  new webpack.ProvidePlugin({
    // 增加对 CodeMirror 的实例的映射
    CodeMirror: "codemirror",
    Chance : "chance",
    $: "jquery",
    jQuery: "jquery",
    "window.jQuery": "jquery",
    "window.$": "jquery"
  })
]
```

以上的配置需要同时加入到 `~/build/webpack.dev.conf.js` 和 `~/build/webpack.prod.conf.js` 这两个配置文件之中。

HtmlEditor 的代码实现

我打算用 Render 方法来实现 HtmlEditor，在 `~/src/components` 下创建 `HtmlEditor.js` 文件，定义 `value` 属性并按照 UIKit 提供的 HtmlEditor 的 HTML 结构在 Render 方法中进行输出，具体代码如下：

```
export default {
  name: 'HtmlEditor',
  props: {
    'value': {
      type: String,
      default: ''
    }
  },
  render (h) {
    return (
      <div>
        <textarea rows="20"
          ref="editor">
          { this.value }
        </textarea>
      </div>
    )
  }
}
```

```

    </div>
  )
}
}

```

接下来就是要在 `this.$ref.editor` 对象上调用 `UIkit` 的 `HtmlEditor` 方法启用 `HtmlEditor` 组件了，这个应用时机应选定在页面元素构建完成，也就是在 `mounted` 钩子内进行处理：

```

mounted () {
  let htmlEditor = window.UIkit.htmleditor(this.$refs.editor, {
    markdown: true,
    mode: 'tab'
  })
},

```

`HtmlEditor` 这个富文本编辑器通过 `CodeMirror` 库实现所见即所得的 HTML 编辑效果，而其真实的文本内容保存到被隐藏起来的 `<textarea>` 元素内，也就是说，当用户进行文本输入时，`CodeMirror` 会相应触发 `<textarea>` 的原生 `oninput` 事件，我们就利用这一点在 `oninput` 中发出 `onchange` 的组件事件，通知父组件 `HtmlEditor` 的内容发生了变化，具体如下所示。

```

mounted () {
  let htmlEditor = window.UIkit.htmleditor(this.$refs.editor, {
    markdown: true,
    mode: 'tab'
  })
  let self = this
  this.$ui.$(this.$refs.editor).on('input', () => {
    const _val = htmlEditor.editor.getValue()
    if (_val !== self.value) {
      this.$emit('change', _val)
    }
  })
},

```

通过测试后，我们将 `HtmlEditor` 应用到数据表单的第二个分页内：

```

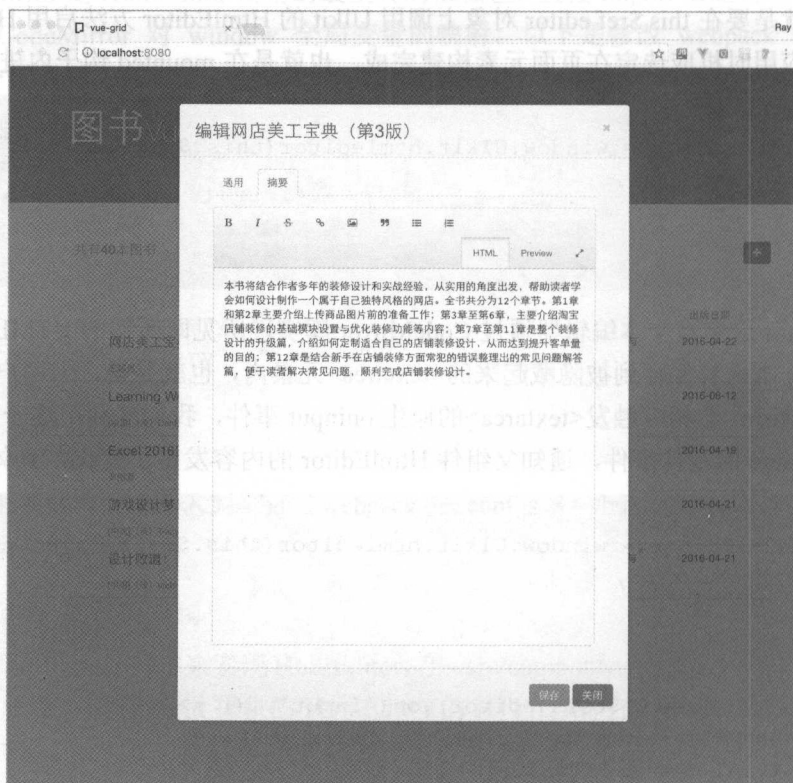
<!--省略-->
<li>
  <!-- 摘要页的内容 -->
  <html-editor :value="current.summary"
    @change="current.summary = $event"></html-editor>
</li>
<!--省略-->

```

`$event` 是 `Vue` 实现的一个全局变量，在事件被执行时得到，我们在 `HtmlEditor` 组件内

用\$emit 发出的 change 事件是将当前编辑内容作为事件参数进行公布的，我们可以直接将其赋值给 current.summary 以模拟自定义组件的双向绑定效果。

最后在浏览器观察 HtmlEditor 的输出是否符合我们的要求：



6.4.3 实现嵌套式容器组件

在一个小小的对话框内如果从上自下地显示所有的表单内容，显然长度不够而且使用体验也不好，我们应该对输入的字段进行分组布局。那么 Tabs 组件必然是首选，因为 Tabs 组件是从 Windows 年代就有且用户不需要学习就知道用法的基本组件。UIKit 也提供 Tabs 的组件效果，其网页结果如下所示。

```
<ul class="uk-tab"
  data-uk-tab="{active:0,connect:'#tabContents'}">
  <li><a href="">通用</a></li>
  <li><a href="">摘要</a></li>
</ul>
```

```
<ul id="tabContents">
  <li> <!-- 通用页的内容 --> </li>
  <li> <!-- 摘要页的内容 --> </li>
</ul>
```

这种结构从理解到记忆上都会显得很差，而且如果我们要在 Vue 里面进行分页的切换也是非常麻烦的，所以应该考虑将它进行组件化。先来设计一下<tabs> 组件使用的接口。我们期望这个组件可以像以下代码这样来使用：

```
<tabs :index="1">
  <tab label="通用">
    <!-- 通用页的内容 -->
  </tab>
  <tab label="摘要">
    <!-- 摘要页的内容 -->
  </tab>
</tabs>
```

在代码中的引入方式应该是这样的：

```
import {Tabs,Tab} from './components/tabs.js'
```

使用接口与引用方式都确定下来，我们就可以动手进行组件的实现了，先在~/src/components/tabs.js 文件内写入以下代码：

```
export const Tab = {
  name : 'Tab'
};

export const Tabs = {
  name : 'Tabs'
};
```

然后编写 Tabs 的单元测试文件，建立~/src/components/tabs.spec.js，将上文中我们所期望的用法写到测试文件中，测试是否能输出 UIKit 的 Tabs 所要求的 HTML。

```
import Vue from 'vue'
import {Tabs, Tab} from './tabs'

describe('tabs', () => {

  describe('tab', () => {

    it('#$mount()', () => {

      let vm = new Vue({
```



```

    el: document.createElement('div'),
    template: `<ul>
      <tab label="通用">
        <div class="tab-content"></div>
      </tab>
      <tab label="摘要">
        <div class="tab-content">ABCDEFG</div>
      </tab>
    </ul>`,
    components: {Tab}

  }).$mount()

  expect(vm.$el.querySelectorAll('li').length).to.equal(2)

  expect(vm.$el.querySelectorAll('.tab-content').length).to.equal(2)
})

})

it('#$mount()', () => {
  let vm = new Vue({
    el: document.createElement('div'),
    template: `<tabs>
      <tab label="通用">
        <div class="tab-content"></div>
      </tab>
      <tab label="摘要">
        <div class="tab-content"></div>
      </tab>
    </tabs>`,
    components: {Tabs, Tab}
  }).$mount()

  expect(vm.$el.querySelectorAll('ul').length).to.equal(2)
  expect(vm.$el.querySelectorAll('a').length).to.equal(2)
  expect(vm.$el.querySelectorAll('.tab-content').length).to.equal(2)
})
});

```

有了测试程序作为引导，我们就有了程序入口与调试的办法。外观是由 UIKit 确定的，只要输出的网页结构没错就一定不会出现显示问题，可以推导出这样的原则：“只要界面被固定后，组件开发完成的标志就是通过单元测试”。当然我们的测试程序必须得覆盖组

件各项的接口与关键点，上述的这条原则才成立。

Tabs 这个组件比较特殊的地方就是它是一个父子式的嵌套型复合组件，而且子组件 `<tab>` 还必须具有容器效果。那么首先来实现子组件 `<tab>`：

```
export const Tab = {
  name: 'Tab',
  props: {
    label: {
      type: String,
      default: 'Tab'
    }
  },
  template: '<li><slot></slot></li>'
};
```

如上述代码所示，`<tab>` 组件就是一个用 `` 标记来容纳自定义内容的容器组件。

接下来就是父容器组件 `<tabs>` 了，我们现在要解决一个问题，就是在 `<tabs>` 中以程序方式取出当前的容器内有多少个 `<tab>`，然后才能获取到 `label` 属性上的值，输出为以下的网页结构：

```
<ul class="uk-tab">
  data-uk-tab="{active:0,connect:'#tabContents'}">
  <li><a href=""><!-- 子组件 tab.label --> </a></li>
  <li><a href=""><!-- 子组件 tab.label --> </a></li>
</ul>
<ul id="tabContents">
  <!--所有子组件的输出-->
</ul>
```

为此我做了多次的研究与尝试，终于找到了获取子组件实例的方法。秘密就在于 `$slots.default` 这个对象。如果我们进入调试模式查看 `Vue` 的内部运行变量值，就会发现 `$slots.default` 是一个可枚举的对象数组，它存的是一堆的 `VNode`。这样一来我们可以在这个组件渲染之前就从中读取数据，那我们可以选择在 `beforeMount` 钩子中做这个处理：

```
export const Tabs = {
  data () {
    return {
      tabItems: []
    }
  },
  beforeMount () {
    this.tabItems = this.$slots.default.filter(node =>
```

```

// 通过名称取出叫 tab 的所有组件
node.componentOptions && node.componentOptions.tag === 'tab'

).map((node, index) => {

  // 读取每个 tab 的索引号
  const data = node.componentOptions.propsData
  data.index = index
  return node
})
}

// ...
};

```

取出了 `tabItems` 的实例就好办了，接下来就是直接制作模板了：

```

export const Tabs = {
  template: `<div>
    <ul class="uk-tab"
      data-uk-tab="{ active:${active}, connect:'#tabContents' }">
      <li v-for="tab in tabItems">
        <a href="">{ tab.label }</a>
      </li>
    </ul>
    <ul class="uk-switcher uk-margin"
      id="tabContents">
      <slot></slot>
    </ul>
  </div>`,

  // ...
}

```

运行测试，通过！这样嵌套式容器组件就宣告完成了。

6.4.4 表单的验证

在 Vue 世界中 `vue-validator` 可能是最流行的表单验证库了，但很可惜的一点是，`vue-validator` 暂时还没有完全支持 Vue2 的发布版本，我曾采用 `vue-validator@3.0.1.alph.1` 的 `alph` 版本（`vue-validator 3`）进行尝试，但在运行期总会报出各种的错误，非常不稳定，

更致命的是这个版本官方还没有提供任何的使用文档，只能期望 `vue-validator 3` 尽快完善和发布正式版本。

除了 `vue-validator`，其实还有另一个替代品那就是 `vee-validate`。`vee-validate` 也是一个不错的选择，入门的学习曲线比较平滑，文档和范例也相当丰富，唯一的缺陷是官方没有提供中文的使用文档。

安装

```
$ npm i vee-validate@next -D
```

加上 (???) 才是支持 Vue2 的版本，`vee-validate` 的默认安装版本是支持 Vue1.x 的。

在入口文件 `main.js` 中应用 `vee-validate` 插件：

```
import Vue in 'vue'
import VeeValidate in 'vee-validate'
Vue.use(VeeValidate)
```

由于 `vee-validate` 的默认语言是英文，所以需要在全局配置中设定区域语言。

```
import Vue from 'vue'
import VeeValidate from 'vee-validate'

Vue.use(VeeValidate)
```

基本用法

`vee-validate` 的使用非常简单，它只要求被验证的输入元素 (`<input>`, `<select>`, `<textarea>`) 声明以下三个基本属性：

- `v-validate`——应用 `vee-validate` 验证；
- `name`——为验证元素声明具体的名称；
- `data-rules`——对输入元素应用具体的验证规则。

以下是它的一个简单示例：

```
<input v-validate
      name="price"
      data-rules="required|numeric|min:20" />
```

验证规则

`data-rules` 的设置方式很容易理解，就像 Vue 的过滤器一样，多个规则之间用 “|” 进行间隔，最后一个值为字段名称，如：

```
const single = 'required' // 单一规则
const multiple = 'required|numeric' // 多个验证规则
```

如果某些验证规则需要输入参数，则在规则名称与参数值之间添加“:”进行分割：

```
const numRule = 'in:1,2,3,4'
```

vee-validate 提供了很丰富的验证规则，具体的内容请参考“附录 B——Vee-Validate 验证规则参考”。

errors 对象

当 vee-validate 完成验证后，会将验证结果保存到 `this.errors` 对象内，我们可以通过该对象判断字段的验证状态，在字段验证后如果出现数据错误，则在输入框的下方显示一段错误的提示文字，我们就能通过 `errors` 控制文字消隐并从 `errors` 对象中提取错误信息：

```
<small class="uk-text-danger"
  v-show="errors.has('pages')">{{errors.first('pages')}}</small>
```

- `errors.has('pages')`——`pages` 字段是否有验证错误；
- `errors.first`——`pages` 验证结果可能有多个（因为一个字段可以加入多个验证规则，如果多个规则都验证失败就会有多条错误信息），`first` 只是显示第一条验证信息的内容。

这个 `errors` 的数据存储结构如下：

```
{
  errors: [
    { field: '字段名', msg: '错误信息' },
    { field: '字段名', msg: '错误信息' },
    { field: '字段名', msg: '错误信息' }
  ]
}
```

这个 `errors` 是 vee-validate 库提供的一个 `ErrorBag` 类型的实例，除了 `first` 方法，它还提供以下对错误信息的处理方法：

- `add(String field, String msg)`——增加一条错误信息到内部列表数组中。
- `all()`——获取全部的验证错误信息。
- `any()`——检测是否有验证错误信息，只要有一条错误信息都会返回真。
- `clear()`——删除全部的验证错误信息。
- `collect(String field)`——返回指定字段名内的全部错误信息，如果不指定则会对全部内容进行分组显示。

- `count()`——获取验证错误信息的总数。
- `first(String field)`——返回指定字段名的第一条错误信息。
- `has(String field)`——检查指定字段名内的所有验证错误。
- `remove(String field)`——删除指定字段名内的所有验证错误信息。

验证的时机

触发 `vee-validate` 进行验证需要调用 `form` 的 `submit`，或者调用 `$validator.validateAll()` 强制进行验证，例如我们希望用户输入 `vee-validate` 后就马上执行验证：

```
<template>
  <form>
    <input name="account"
      @input="handleValidate"
      data-rules="required|email" />
  </form>
</template>
<script>
  export default {
    methods: {
      handleValidate (e) {
        this.$validator.validateAll()
      }
    }
  }
</script>
```

但如果希望使用更简单直接的方法，令输入组件加载后就自动执行验证的话，只要向 `v-validate` 指令标记指定要验证的字段名表单就会在加载后自动验证，具体如下所示。

```
<input name="email" v-validate="email" data-rules="required" />
```

验证消息的本地化

`vee-validate` 是将验证的消息的描述内容进行了标准化，针对每一种验证规则就可以得到一致化的提示信息，这样一来当我们要开发大量具有验证规则的表单时，就不会重复地编写类似的验证消息内容，这对于交互界面的一致化有着很大的益处。`vee-validate` 默认是采用英文消息模板的，所以我们在初始化时需要将中文消息在全局设定中引入：

```
import Vue from 'vue'
import messages from 'vee-validate/dist/locale/zh_CN'
import VeeValidate from 'vee-validate'
```



```
Vue.use(VeeValidate, {
  locale: 'zh_CN',
  dictionary: {
    zh_CN: {messages}
  }
})
```

上述代码中的 `messages` 对象相当于一个表单验证专用的中文字典模板，为了可以更清楚理解它的工作机制，打开这个字典来一探究竟：

```
export default {
  after: (field, [target]) => ` ${field} 必须在 ${target} 之后 `，
  alpha_dash: (field) => ` ${field} 能够包含字母数字字符，包括破折号、下划线 `，
  alpha_num: (field) => ` ${field} 只能包含字母数字字符 `，
  alpha_spaces: (field) => ` ${field} 只能包含字母字符，包括空格 `，
  alpha: (field) => ` ${field} 只能包含字母字符 `，
  before: (field, [target]) => ` ${field} 必须在 ${target} 之前 `，
  between: (field, [min, max]) => ` ${field} 必须在 ${min} ${max} 之间 `，
  confirmed: (field, [confirmedField]) => ` ${field} 不能和 ${confirmedField}
  匹配 `，
  date_between: (field, [min, max]) => ` ${field} 必须在 ${min} 和 ${max} 之间 `，
  date_format: (field, [format]) => ` ${field} 必须在 ${format} 格式中 `，
  decimal: (field, [decimals] = ['*']) => ` ${field} 必须是数字的而且能够包含
  ${decimals} === '*' ? '' : decimals} 小数点 `，
  digits: (field, [length]) => ` ${field} 必须是数字，且精确到 ${length} 数 `，
  dimensions: (field, [width, height]) => ` ${field} 必须是 ${width} 像素到
  ${height} 像素 `，
  email: (field) => ` ${field} 必须是有效的邮箱 `，
  ext: (field) => ` ${field} 必须是有效的文件 `，
  image: (field) => ` ${field} 必须是图片 `，
  in: (field) => ` ${field} 必须是一个有效值 `，
  ip: (field) => ` ${field} 必须是一个有效的地址 `，
  max: (field, [length]) => ` ${field} 不能大于 ${length} 字符 `，
  mimes: (field) => ` ${field} 必须是有效的文件类型 `，
  min: (field, [length]) => ` ${field} 必须至少有 ${length} 字符 `，
  not_in: (field) => ` ${field} 必须是一个有效值 `，
  numeric: (field) => ` ${field} 只能包含数字字符 `，
  regex: (field) => ` ${field} 格式无效 `，
  required: (field) => ` ${field} 是必需的 `，
  size: (field, [size]) => ` ${field} 必须小于 ${size} KB `，
  url: (field) => ` ${field} 不是有效的 url `
}
```

vee-validate 预载了 14 个不同国家和地区的验证规则语言字典。从代码中就了解到其实

质上是对应于验证规则的格式化字符串而已。field 参数就是被验证的字段，默认情况下 vee-validate 会将字段的名称直接用于格式化，这种情况是我们不希望看到的（至少非英文程序是这样的），所以我们可以使用一个 data-as 属性，用中文别名取代英文字段名进行格式化的方式如下：

```
<input id="authors"
  name="authors"
  :class="{
    'uk-form-width-large':true,
    'uk-form-danger': errors.has('authors')
  }"
  v-validate
  data-rules="required|authors"
  v-model="authors"
  data-as="作者" />
```

这样我们的验证消息就不会出现中英文混杂的情况了。

验证器实例一次只能生成一种语言的提示消息。必要时可以使用 setLocale 方法来切换验证器使用的语言：validator.setLocale('zh_CN')。

讲了这么多关于 vee-validate 的基础理论，也该是将它应用到我们的表单中的时候了，以下是应用了表单验证的完整代码：

```
<tabs>
  <tab label="通用">
    <div class="uk-form-row">
      <label class="uk-form-label"
        for="name">书名</label>
      <div class="uk-form-controls">
        <input id="name"
          name="name"
          :class="{
            'uk-form-width-large':true,
            'uk-form-danger': errors.has('name')
          }"
          autofocus="autofocus"
          @input="handleValidate"
          v-validate
          data-rules="required"
          data-as="书名"
          v-model="current.name"/>
        <small class="uk-text-danger"
          v-show="errors.has('name')">{{errors.first('name')}}</small>
```

```

    </div>
  </div>
  <div class="uk-form-row">
    <label class="uk-form-label"
      for="isbn">书号</label>
    <div class="uk-form-controls">
      <input id="isbn"
        name="isbn"
        :class="{
          'uk-form-width-large':true,
          'uk-form-danger': errors.has('isbn')
        }"
        @input="handleValidate"
        v-validate
        data-rules="required"
        data-as="书号"
        v-model="current.isbn"/>
      <small class="uk-text-danger" v-show="errors.has('isbn')">{{errors.
first('isbn')}}</small>
    </div>
  </div>
  <div class="uk-form-row">
    <label class="uk-form-label"
      for="price">售价</label>
    <div class="uk-form-controls">
      <input id="price"
        name="price"
        @input="handleValidate"
        :class="{
          'uk-form-width-large':true,
          'uk-form-danger': errors.has('price')
        }"
        type="number"
        v-validate
        data-rules="required|numeric|min:0"
        data-as="售价"
        v-model="current.price"/>
      <small class="uk-text-danger" v-show="errors.has('price')">{{errors.
first('price')}}</small>
    </div>
  </div>
  <div class="uk-form-row">
    <label class="uk-form-label"
      for="published">类别</label>

```

```

<div class="uk-form-controls">
  <input id="category"
    type="date"
    :class="{
      'uk-form-width-large':true,
      'uk-form-danger': errors.has('category')
    }"
    name="category"
    @input="handleValidate"
    data-as="类别"
    v-model="current.category"
    v-validate
    placeholder="YYYY-MM-DD"
    data-rules="required" />
    <small class="uk-text-danger" v-show="errors.has('category')">
      {{errors.first('category')}}</small>
</div>
</div>
<div class="uk-form-row">
  <label class="uk-form-label"
    for="published">出版日期</label>
  <div class="uk-form-controls">
    <input id="published"
      type="date"
      :class="{
        'uk-form-width-large':true,
        'uk-form-danger': errors.has('published')
      }"
      @input="handleValidate"
      name="published"
      v-model="current.published"
      data-as="出版日期"
      v-validate
      placeholder="YYYY-MM-DD"
      data-rules="date_format:YYYY-MM-DD"
    />
    <small class="uk-text-danger" v-show="errors.has('published')">
      {{errors.first('published')}}</small>
  </div>
</div>
<div class="uk-form-row">
  <label class="uk-form-label">&nbsp;</label>
  <div class="uk-form-controls">
    <label>
      <input type="checkbox"

```



```

        v-model="is_published"/> 上市销售
    </label>
</div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="pages">页数</label>
    <div class="uk-form-controls">
        <input id="pages"
            name="pages"
            :class="{
                'uk-form-width-large':true,
                'uk-form-danger': errors.has('pages')
            }"
            type="number"
            step="any"
            @input="handleValidate"
            v-validate
            data-as="页数"
            data-rules="numeric|min:20"
            v-model="current.pages"/>
        <small class="uk-text-danger" v-show="errors.has('pages')">{{errors.
first('pages')}}</small>
    </div>
</div>
<div class="uk-form-row">
    <label class="uk-form-label"
        for="authors">作者</label>
    <div class="uk-form-controls">
        <input id="authors"
            name="authors"
            :class="{
                'uk-form-width-large':true,
                'uk-form-danger': errors.has('authors')
            }"
            @input="handleValidate"
            v-validate
            data-rules="required"
            v-model="authors"/>
        <small class="uk-text-danger"
            v-show="errors.has('authors')">{{errors.first('authors')}}
</small>
    </div>
</div>

```



```

</tab>
<tab label="摘要">
  <html-editor :value="current.summary"
    @change="onSummaryChange"></html-editor>
</tab>
</tabs>
<!--省略-->
<script>
export default {
  methods: {
    onSummaryChange (val) {
      current.summary = val
      this.handleValidate()
    },
    handleValidate (e) {
      this.$validator.validateAll()
    },
    // ... 省略
  }
}
</script>

```

虽然 `vee-validate` 的学习曲线相比平缓, 学起来感觉会很容易, 但实际写起来还是有点费劲, 每增加一个字段大约就需要 10 多行的代码。而且这个表单在逻辑上具有很强的重复性。消除不必要的重复是组件化开发的一项非常必要且持久的工作, 分析上述加入数据验证后的代码就会发现所有的输入项具有以下的编写规则:

```

<div class="uk-form-row">
  <label class="uk-form-label"
    for="字段名">字段标题</label>
  <div class="uk-form-controls">
    <input id="字段名"
      name="字段名"
      :class="{
        'uk-form-width-large':true,
        'uk-form-danger': errors.has('字段名')
      }"
      type="字段类型"
      @input="handleValidate"
      v-validate
      data-as="字段标题"
      data-rules="numeric|min:20"
      v-model="current.pages"/>
    <small class="uk-text-danger" v-show="errors.has('字段名')">{{errors.

```

```
first('字段名')}}</small>
</div>
</div>
```

只要能发现重复规律就可以进行抽象与封装，继续沿用我一直在讲述的封装组件的思路，先在测试中写出封装后的组件是如何使用的，在编写这个假定组件的同时不断地简化它的用法直至无法再简化为止。现在我们假定一个 `form-field` 组件来简化上述代码：

```
<form-field name="字段名称"
  label="字段标题"
  rules="验证规则"
  :value="目标绑定字段"
  input-type="字段类型">
</form-field>
```

具体测试代码如下：

```
describe('FormField', () => {
  it('#mount', () => {
    let vm = new Vue({
      el: document.createElement('div'),
      render(h) {
        return <div>
          <form-field label="姓名"
            name="name"
            rules="require"
            value="Ray">
          </form-field>
        </div>
      },
      components: {FormField}
    }).$mount()

    expect(vm.$el.querySelector('label').textContent).to.eqls('姓名')
    expect(vm.$el.querySelector('label').getAttribute('for')).to.
    eqls('name')
    expect(vm.$el.querySelector('input').value).to.eqls('Ray')
  })
})
```

按照上述的设计逻辑实现 `FormField` 组件：

```
export default {
  name: 'FormField',
  props: {
    name: {type: String},
```

```

    label: {type: String},
    rules: {type: String},
    value: {},
    inputType: {type: String, default: 'text'}
  },
  render (h) {
    return <div class="uk-form-row">
      <label class="uk-form-label"
        for={this.name}>{this.label}</label>
      <div class="uk-form-controls">
        <input id={this.name}
          name={this.name}
          type={this.inputType}
          class={{
            'uk-form-width-large': true
          }}
          value={this.value}/>
      </div>
    </div>
  }
}

```

现在 `FormField` 并没有应用验证器，此时我们将使用 `VeeValidator` 的另一种更高级的用法，就是通过 JS 代码控制验证过程。

首先，当调用 `Vue.use(VeeValidate)` 后，`Vee` 验证器会自动将 `$validator` 注入到 `Vue` 实例中。它是一个独立的类，可以单独用来编写验证值，这种验证器引用方式属于全局性的，这也是前文提及的内容。但 `FormField` 是一个独立的组件，如果我们没有在 `main.js` 的全局文件内注册验证器，可能会导致引用失败，所以我们需要在 `FormField` 中查找当前可用的验证器实例，如果没有找到的话还可以构造一个。

在动手前先来看看如何手动构造一个验证器实例，第一种方式是用构造函数在构造验证器的同时输入验证规则：

```

import { Validator } from 'vee-validate';
const validator = new Validator({
  email: 'required|email',
  name: 'required|alpha|min:3',
});

```

另一种方式就是使用无参的构造函数创建 `Validator` 实例，以后再添加验证规则。

```

import { Validator } from 'vee-validate';
const validator = new Validator()

```

```

validator.attach('email', 'required|email') // 只附加验证规则
validator.attach('name', 'required|alpha', '全名') // 附加验证规则的同时输入验证失败时出错的信息，相当于 data-as
validator.detach('email') // 还可以将附加的规则移除

```

之后，可以使用 `validate(field, value)` 验证值，如果所有验证通过，则应返回布尔值，就像这样：

```

validator.validate('email', 'foo@bar.com'); // true
validator.validate('email', 'foo@bar'); // false

```

当我们了解了这些基本的用法后，就可以向 `FormField` 中加入验证规则了，现在还得确定的一点那就是验证的时机，我们设计这个 `FormField` 是在输入时自动进行验证，那么这个验证时机就设定在 `input` 元素的 `on-input` 事件来进行触发，代码如下：

```

import {Validator} from 'vee-validate'

export default {
  name: 'FormField',
  props: {
    name: {type: String},
    label: {type: String},
    rules: {type: String},
    value: {},
    inputType: {type: String, default: 'text'}
  },
  mounted () {
    if (this.rules)
      this.validator.attach(this.name, this.rules)
  },
  methods: {
    onValidate (e) {
      if (this.rules) {
        this.validator.validate(this.name, e.target.value)
      }
    }
  },
  computed: {
    validator () {
      if (this.$parent.$validator)
        return this.$parent.$validator
      else {
        if (this._validator)
          return this._validator

```



```

    else {
      const validator = new Validator()
      validator.attach(this.name, this.rules, this.label)
      this._validator = validator
      return validator
    }
  },
  errorBag () {
    return this.validator.errorBag
  },
  hasError () {
    return this.errorBag.has(this.name)
  },
  errorMsg () {
    this.errorBag.first(this.name)
  }
},
render (h) {
  return <div class="uk-form-row">
    <label class="uk-form-label"
      for={this.name}>{this.label}</label>
    <div class="uk-form-controls">
      <input id={this.name}
        name={this.name}
        type={this.inputType}
        class={{
          'uk-form-width-large': true,
          'uk-form-danger': this.hasError
        }}
        on-input={ this.onValidate }
        value={this.value}/>
      <small class={{
        'uk-text-danger': true,
        'uk-show': this.hasError
      }}>{ this.errorMsg }</small>
    </div>
  </div>
}
}

```

所有的验证错误都会被保存到 `ErrorBag` 对象中，所以我们可以直接访问验证器的 `errorBag` 属性或者使用 `getErrors()` 进行读取：

```

errorBag () {
  return this.validator.errorBag
}

```



```

},
hasError () {
  return this.errorBag.has(this.name)
},
errorMsg () {
  this.errorBag.first(this.name)
}

```

此时有了 `FormField` 这个组件，那么我们就可以将所有的字段信息提取为一个数组，然后用 `v-for` 统一输出所有的输入字段：

```

<template>
  <div>
    <form class="uk-form uk-form-horizontal"
      v-if="current">
      <div class="uk-container uk-container-center">
        <ul class="uk-tab" data-uk-tab="{active:0,connect:'#tabContents'}">
          <li><a href="">通用</a></li>
          <li><a href="">摘要</a></li>
        </ul>
        <ul id="tabContents">
          <li>
            <form-field v-for="field in fields"
              :name="field.name"
              label="field.label"
              rules="field.rules"
              input-type="field.type"
              :value="current[field.name]">
            </form-field>
          </li>
          <li>
            <!-- 摘要页的内容 -->
            <html-editor :value="current.summary"
              @change="current.summary = $event"></html-editor>
          </li>
        </ul>
      </div>
    </form>
  </div>
</template>
<script>
  import HtmlEditor from './htmleditor'
  import FormField from './form-field'
  import {Tab, Tabs} from './tabs'
  export default {

```

```

    props: ['book'],
    data () {
      return {
        current: {},
        fields: [
          {name: 'name', label: '书名', rules: 'required', type: 'text'},
          {name: 'isbn', label: '书号', rules: 'required', type: 'text'},
          {name: 'price', label: '售价', rules: 'required|numeric|min:0',
type: 'number'},
          {name: 'category', label: '类别', rules: 'required', type: 'text'},
          {name: 'published', label: '出版日期', rules: 'required', type:
'date'},
          {name: 'pages', label: '页数', rules: 'required|numeric|min:20',
type: 'number'},
          {name: 'authors', label: '作者', rules: 'required', type: 'text'}
        ]
      }
    },
    components: {
      HtmlEditor, FormField, Tab, Tabs
    }
  }
</script>

```

这样一次组件化的重构仍然使用的是第4章中介绍的组件化思路，将接近200行的代码“浓缩”到仅仅只有50行左右！

6.5 集成服务端的 CRUD Restful API

前面我们为了能快速实现 CRUD 的操作逻辑，暂时没有引入过多的复杂性，直接从 JSON 文件中将样本数据读入到 books 变量内，现在是时候将其重构为与服务器通信的真实功能了。首先，我们要进行一些基本知识的学习，深入了解一下 vue-resource 的另一个高层次对象 resource，使用它会大量地减少很多不必要的重复性代码的出现，既增强了程序的封装程度，又提高了代码的可读性。

vue-resource 是 Vue.js 的一款插件，它可以通过 XMLHttpRequest 或 JSONP 发起请求并处理响应。也就是说，\$.ajax 能做的事情，vue-resource 插件一样也能做到，而且 vue-resource 的 API 更为简洁。另外，vue-resource 还提供了非常有用的 inteceptor 功能，使用 inteceptor 可以在请求前和请求后附加一些行为，比如使用 inteceptor 在 AJAX 请求时显示 loading 界面。

当我们对后端同一个数据对象的 CRUD 操作时，实质上都是对同一个 URL 进行操作，区别只是 HTTP 方法有所差异，这是 RESTful API 的实现通则或者说是一种使用习惯。以下是将本示例中的图书对象作为资源名称，从 HTTP 协议上来看看 RESTful CRUD API，如下表所示。

HTTP 请求	说 明
HTTP GET /api/books/	获取所有的图书
HTTP POST /api/books/	创建新的图书数据项
HTTP PUT /api/books/:id	更新指定的图书对象 (:id 为图书的唯一编号)
HTTP DELETE /api/books/:id	删除指定 id 的图书

如果对每次调用都直接采用 \$http 对象来包装以上的 CRUD 逻辑，除了 HTTP 方法与返回值有一点差异，其他的实现逻辑几乎都是重复的。为了简化这一操作，vue-resource 提供了 resource 对象，resource 对象对 \$http 对象在常规 CRUD 的操作都进行了包装，使用时只需要对 URL 进行一次初始化，将这个 URL 视为一个“资源”对象，调用这个“资源”对象上的操作方法就可以产生对应的 HTTP 请求。

resource 的实例化方法如下：

```
resource(url, [params], [actions], [options])
```

resource 对象支持的方法如下：

- get: {method: 'GET'}——读取单条数据；
- save: {method: 'POST'}——创建新的数据项；
- query: {method: 'GET'}——获取多条数据项；
- update: {method: 'PUT'}——更新指定数据项；
- remove: {method: 'DELETE'}——删除指定数据项；
- delete: {method: 'DELETE'}——与 remove 相同，是一个别名。

简言之，resource 就是对 \$http 进行标准化 CRUD 包装与抽象的更高层级的对象。当然这还需要服务器端对 CRUD 的 REST API 提供支持。

接下来就将 vue-resource 植入到代码内，这里会比之前的实现逻辑再复杂一些——加入了分页的功能，没有分页的 CRUD 是不完整的，真实的使用体验也是非常差的。我们可以选择在页面构造的时候就开始从服务端加载数据，在 created 钩子方法中加入以下代码：

```
created () {  
  this.bookService = this.$resource('/api/books')  
  this.fetchBooks()  
}
```

`this.bookService` 只是一个新构造出来的 `resource` 对象，实质上并没有进行任何的处理，`fetchBooks()` 方法才真正调用 `resource` 获取数据。`fetchBooks` 的实现如下：

```
data () {
  books: [],
  totalBooks : 0,
  currentPage: 0,
  pageSize: 5
  // ..
},
methods: {
  fetchBooks (options) {
    var defaults = {
      pageIndex: this.currentPage,
      pageSize: this.pageSize
      filter: this.terms,
      sort: this.sortingKey,
      dir: this.direction
    }

    return this.bookService
      .query(options ? _.extend({}, defaults, options) : defaults)
      .then((res) => {
        this.books = res.body.data
        this.totalBooks = res.body.total_records
      }, (error) => {
        console.log(error)
      })
  },
  // ...
}
```

`fetchBooks` 是一个很有可能要重构多次输入参数的方法，而且这个方法会被多次引用，因此为了不造成接口的变化，我们应该将输入参数对象化，使用固定默认值对象 `defaults`。上述代码中 `totalBooks`、`currentPage` 和 `pageSize` 这些属性是为了支持分页加入的，可以作为默认值使用。这样 `fetchBooks` 就具备了刷新数据的功能，如果输入参数 `options` 有值，则使用 `_.extend` 将新值重写成默认值，这样无论参数如何增加都不会造成原有代码变更了。

接下来是排序功能的重构，服务端排序就比客户端排序在逻辑上简单多了：

```
sortBy (key) {
  if (key === this.sortingKey) {
    this.direction = this.direction === 'asc' ? 'desc' : 'asc'
  }
  this.sortingKey = key
}
```



```

    this.fetchBooks()
    // this.books = _.orderBy(this.books, key, this.direction)
  }

```

只是将原有的客户端排序 `_orderBy` 方法换成 `fetchBooks`。这就是 CRUD 的 **R(Read)** 动作，我们这里是隐式实现单个读取，读取的是一批数据而不是一个，所以使用 `resource.query` 这个方法会将输入的参数转化为经过编码的完整带参 URL，然后向服务端发出 GET 请求。

进行添加与更新操作时要注意的一点是，执行 `resource.update` 时如果对象没有 `id` 属性，就需要在 `update` 方法中明确指定，这样才能生成正确的 URL，具体代码如下：

```

save () {
  if (this.isEditing) {
    this.bookService.update({ id: this.current.isbn }, this.current)
  } else {
    this.bookService.save(this.current)
  }

  this.fetchBooks()
}

```

最后就是删除 (D) 操作了，本示例支持多项同时删除的功能，所以这里并不会调用 `resource.delete`，而采用 `$http` 对象来进行操作，代码如下：

```

removeBooks () {
  // $ui.confirm 是由 UIKit 提供的标准方法，显示一个确认对话框
  this.$ui.confirm('真的要删除所选中的图书吗?', () => {
    // 将要删除的 selection 数组作为对象写入到 HTTP body 内
    this.$http.delete('/api/books', {body: this.selection})
      .then((res) => {
        this.selection = []
        this.refreshBooks()
      })
  })
}

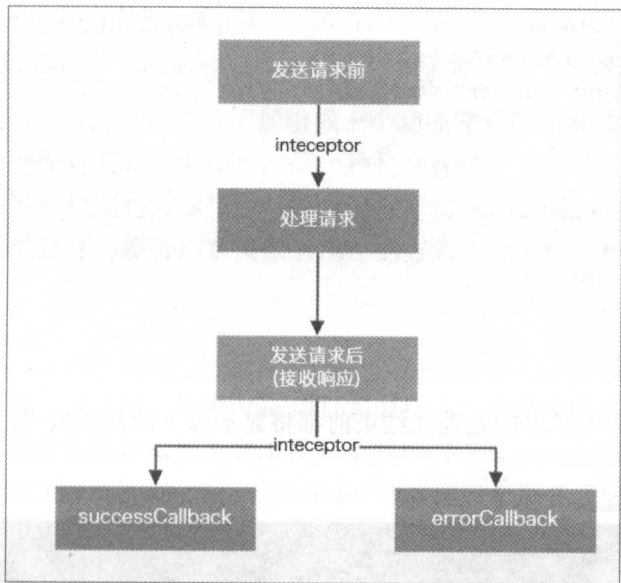
```

6.6 HTTP 拦截器 inteceptor

上述的 CURD 示例有一处用户体验不太好，就是用户在调用 `resource` 的时候如果网络较慢，画面又没有给出反馈，用户不知道操作是成功还是失败的，也不知道是否该继续等

待。这个时候我们应该用拦截器在请求发送前和发送请求后做一些处理，显示一个加载对话框。

先看看拦截器的工作原理：



拦截器的基本用法

```
Vue.http.interceptors.push((request, next) => {  
  // ...  
  // 请求发送前的处理逻辑  
  // ...  
  next((response) => {  
    // ...  
    // 请求发送后的处理逻辑  
    // ...  
    // 根据请求的状态，response 参数会返回给 successCallback 或 errorCallback  
    return response  
  })  
})
```

拦截器是一个全局性的处理，使用它的坏处是所有向服务器发出的请求都进行统一处理，所以只能显示泛用性的提示，这是一个需要在应用时考虑的实际问题。如果要显示有针对性的提示，那么请考虑在 then 方法或者 error 处理器内进行独立的操作。

当与服务器进行通信时，我们显示一个“加载中...”的拦截画面隔绝用户进行其他的操作，待处理完成后才删除加载画面。可以使用前文制作 Vue 插件时植入的 blockUI 方法

实现。另外我们要将拦截器的代码写在 `main.js` 文件内，因为它是一个全局性方法，具体代码如下所示。

```
Vue.http.interceptors.push((request, next) => {  
  if (request.url.indexOf('/api/') > -1) {  
    let modal = window.UIkit.modal.blockUI(`<div  
class="uk-modal-spinner"></div>  
<p class="uk-text-center">玩命加载中...</p>`, {  
      center: true  
    })  
  
    next((response) => {  
      modal.hide()  
      return response  
    })  
  }  
})
```

这样每个向服务端发出的处理在结束前都将显示以下的加载效果：



6.7 开发服务器的定制

第2章“工程化的 Vue.js 开发”中曾提到过，vue-cli 的 webpack 模板在初始化时就已经预置了一个 Node.js 中最出名也是最常用的 Express 作为 Vue 开发运行环境的服务器，它主要是提供热加载与静态资源（JS 脚本、CSS 样式表和字体等文件）运行时的宿主环境。

众所周知，前端与后端的有机结合才能组成一个功能完整的网站或 Web 应用，在实际的开发中，我们经常会面对两种开发场景：第一种是纯粹只开发前端项目，集成已有的成熟的后端服务资源；另一种则是前端与后端同时进行开发。无论遇到的是哪一种，我们都不可避免地要在 Vue 的程序与服务端直接以 AJAX 的方式进行通信，在前端调用后端所提供的服务。

后端服务代理

首先来详细说说第一种情况，如果 Vue 项目需要接入一个现有的后端服务，直接接入的话，由于前端运行的程序与后端服务并不在一个网域内，有可能会被浏览器的安全程序阻止，例如我们要调用 `http://api.demo.com/get/user/:id` 这样一个服务。如前文所述，我们会这样来编写客户端的代码：

```
this.$http.get('/get/user',{id:id})
  .then(res=> {
    // 执行返回后的操作
  })
```

当运行 `$ npm run dev` 的时候，Vue 开发环境实际上是将程序“Host”在 `http://localhost:3000` 下的，这样前端程序就无法正确定位服务了（当然这种情况是出现在服务端与客户端都“Host”在同一个网域的时候），为了避免发生以上这种情况，我们可以在开发服务器上做一个服务代理，将从 Vue 发送到 Express 开发服务器上的 HTTP 请求转发到正确的地址上。

幸好 Vue 官方早就未雨绸缪地为开发者想到了这一点！在 `dev-server.js` 内配置了这样的代理服务，它就是 `http-proxy-middleware` 中间件。这个中间件专门负责对请求进行代理。它实现的效果是这样的，当服务端收到一个指定的请求时，例如以 `/api` 开头的任意地址，我们可以将这个请求转发到一个外部地址，待请求执行完成再将结果返回给客户端。

启用这个代理就相当简单了，在 `config/index.js` 文件内有一个 `proxyTable` 的配置属性，我们将要进行代理的源地址与目标地址关联起来就可以工作了，具体做法如下：

```
// config/index.js
module.exports = {
```

```

build: {
  // ... 省略
},
dev: {
  // ... 省略
  proxyTable: {
    '/api' : { // 本机开发服务器的地址
      target : 'http://api.demo.com', // 后端服务器的地址
      changeOrigin: true
    }
  }
}
}

```

上述方式的意思就是所有以/api 开始的请求都代理至 `http://dev.domain.com/api` 下，`proxyTable` 的键值（key）就是要进行代理的本地地址的匹配模式，它可以有以下用法：

- `**`——匹配所有的路径，全部的请求都会被代理（当然在 Vue 中不要这样做，否则是不能启动的）；
- `**/*.html`——匹配任意以.html 结尾的请求；
- `/*.html`——仅匹配根路径下的任意以.html 结尾的请求；
- `/api/**/*.html`——匹配/api 路径下任意以.html 结尾的请求；
-

这些配置规则是基于 `micromatch` (<https://www.npmjs.com/package/micromatch>) 开发的，更高级的配置写法可以参考它的网站。

代理的配置选项是 `options`，这个 `options` 可以有以下的具体值：

属 性	类 型	说 明
target	string	目标代理地址
forward	string	目标代理地址
agent	Object	HTTP(s)请求头内的 agent 对象，用于设置浏览器相关参数，具体可参考 https://nodejs.org/api/http.html#http_class_http_agent
ssl	Object	传递至 <code>https.createServer()</code> 的 SSL 参数
ws	boolean	如果希望使用 Websockets 代理
xfwd	boolean	在请求内加 x-forward 头信息
secure	boolean	是否使用 SSL 证书验证
toProxy	boolean	是否将地址作为路径方式转发给其他的代理（使用中继代理模式）
prependPath	boolean	声明是否将目标路径插入至代理路径之前，默认值为 true

续表

属 性	类 型	说 明
ignorePath	boolean	true/false, 默认: false——指定是否要忽略传入请求的代理路径(注意: 如果需要, 必须附加/手动)
localAddress	string	指定与外部连接绑定的本机地址
changeOrigin	boolean	true/false, 默认: false——将原主机头更改为目标 URL
auth	string	基本验证 例如: 'user:password' 就会自动计算用于基本验证的请求头
hostRewrite	string	当遇到重定向请求(301/302/307/308)时重写的主机头。
autoRewrite	string	当遇到重定向请求(301/302/307/308)时重写主机头与端口, 默认为 false
protocolRewrite	string	当遇到重定向请求(301/302/307/308)时自动将传输协议重写为 http 或 https, 默认为 null
cookieDomainRewrite	string/Object	重写 cookie 的域
headers	Object	增加一个请求头(例如: {host:'www.example.org'})
proxyTimeout	Number	设置代理超时的时长(以毫秒计算)

cookieDomainRewrite 可以是以下的值:

- false (默认)——禁止重写 cookie。
- String——写入新的主域。例如: cookieDomainRewrite: "new.domain", 要删除域可以用 cookieDomainRewrite: ""。
- Object——将已有域映射至指定的新域名上。使用 "*" 通配符可以匹配全部的域名。例如: 只保持一个主域不被重写, 重写一个域并删除一个域: cookieDomainRewrite: { "unchanged.domain": "unchanged.domain", "old.domain": "new.domain", "*" : "" }。

http-proxy-middleware 还有更多更深入的配置方式以适应各种的应用需要, 具体的内容请参考: <https://github.com/chimurai/http-proxy-middleware>。

除非你的服务端并不是运行在一个生产环境中, 又或者服务端提供了一个可任意重置的开发环境, 否则是不应该在 Vue 项目的开发环境中直接接入到生产环境来进行开发的。开发过程需要进行很多的测试, 开发环境下产生的很多数据或者操作有可能导致生产环境的数据产生混乱。

后端服务仿真

要出现上述效果, 现在的代码还不行, 因为现在还没有接收 API 请求的服务端! 那要等待服务端开发完成才能进行下一步的开发调试吗? 答案显然是否定的, 因为我们可以做服务模拟 (Service Mocking), 也就是直接模拟出一个伪服务端, 在接收 API 请求后输出我们所要求 API 的结果。vue-cli 的 webpack 模板早就为我们准备好了这个模拟环境, 只需要

稍微做一点点的修改。

具体做法有两种，一种是直接开发模拟 API，另一种是用代理服务器将前端发出的 API 请求转发到一个已有的服务端上处理。后者适用于已有服务端实现的情况下，或者是将 API 发到另一个非 Express.js 的服务端上。

先来具体说说第一种做法，在第 2 章我们已经知道 build/dev-server.js 就是开发环境的运行脚本，开发服务器实质上是一个配置很简单的 Express.js 服务端，那么我们就可以利用 Express.js 的中间件特性向 Express.js 实例中插入一个模拟服务端处理的 RESTful API 中间件，在 build 目录下加入一个 mocks.js 文件，具体实现代码如下：

```
var express = require('express');
var router = express.Router();
var mockData = require('../src/fixtures/items.json');
var _ = require('lodash');

router.get('/books', function (req, res) {
  // 查询图书数据
  var page = req.query.page || 0,
      size = req.query.size || 10,
      filter = req.query.filter,
      skipStart = (page > 0 ? page - 1 : page) * size,

  // 筛选符合条件的数据
  filteredData = filter ? _.filter(mockData, function (d) {
    return d.name.indexOf(filter) > -1
  }) : mockData;

  // 对数据进行分页
  pageData = _.take(_.slice(filteredData, skipStart), size);

  res.json({
    total_records: filteredData.length,
    total_pages: Math.ceil(filteredData.length / size),
    page: page,
    size: size,
    data: pageData
  });
})

.post('/books', function (req, res) {
  // 添加新的图书
  mockData.push(req.body)
```

```

    res.json(req.body)
  })
  .put('/books/:isbn', function (req, res) {
    // 更新指定的图书对象
    mockData = _.reject(mockData, function (d) {
      return d.isbn === req.body.isbn
    })
    mockData.push(req.body)
    res.json({msg: 'success'})
  })
  .delete('/books', function (req, res) {
    // 批量删除图书
    mockData = _.reject(mockData, function (d) {
      return _.indexOf(req.body, d.isbn) > -1
    })
    res.json({msg: 'success'});
  });

module.exports = router;

```

这个中间件页就是将原来的数据样本加载到内存中，模拟出了一个内存数据库，Vue 前端的所有操作这个服务端都能处理。

由于 Express.js 的具体内容已经超出了本书的范围，没有接触过 Express.js 服务端开发的读者可以访问 <http://expressjs.com/>，快速地了解一下这个 Node.js 下应用得最广泛同时又是极为轻量级的 Web 服务器的实现，它的学习曲线极为平滑，做过服务端开发的朋友基本上可以在一两个小时内掌握其中的要领。

打开 dev-server.js 文件，引入 mocks 并在 app 实例中挂载这个中间件：

```

var mocks = require('./mocks')
var bodyParser = require('body-parser')
var express = require('express')
// ... 省略
var app = express()
// ... 省略

// 一定要加载 bodyParser，否则不能处理 BODY 中的 JSON
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended: false}));
app.use('/api', mocks)

```

将 mocks.js 的服务端模拟程序挂入后就可以正常启动查看实际运行效果了。

第 7 章 Vuex 状态管理

在 Vue 的生态圈内提供了一个叫 Vuex 的库，专门用于状态共享与状态管理，感觉上它就像 Flux 的复制品。在最初接触 Vuex 的时候我并不喜欢它，虽然从概念图谱上来讲很容易理解，但对于 Vue 它却有着非凡的意义。

开始学习时，即使看着官方提供的中文文档都会感到一脸的茫然，或者是我过于愚钝很难一时间理解它的意义所在，所以就将其弃之一角。使用 Vue 进行项目开发一段时间后，慢慢碰到以下两种问题：

首先，从前几章中我们已经非常清楚地知道 Vue 是一种完全基于组件化的可视化开发框架，它的美在于我们会很自然地用设计原则来规划组件粒度与关系，用面向对象的思维来思考与编程。然而事物都有两面性，当我们将视角拉近一点，关注那些流动在组件与组件之间的变量，尤其是那些较为复杂的复合型组件，会发觉父组件与子组件之间、子代组件与其子组件之间的变量维护变得越来越不容易，由于 Vue 取消了属性的变量同步功能，父子组件之间的变量传递实际上是单向的：当子组件需要对传入的变量进行修改，同时又希望通知父组件传入的变量发生了变化时，则不得不使用事件冒泡来进行这种传递，最后在父组件中使用一个方法来进行重绘或者执行其他相应的行为，此类同质性操作又不能被封装为可重用的代码，这样会不断地加大父组件与子组件之间的耦合度，让它们之间的关系变得错综复杂。

其次，vue-router 可能是绝大多数的 Vue 项目都必不可少的工具库，有了 vue-router，我们可以将一个组件视为一个页面来使用。由于组件只维护自身的状态（data），组件创建时或者说进入路由时它们被初始化，切换至其他的组件页时当前页自然要被销毁，从而导致 data 也随之销毁。页面与页面之间总会产生各种需要的共享变量，如果通过 \$router.param 或者 \$router.meta 来传递是远远不够的，很多情况下不得不采用 window 来保存一些全局的共享变量（有很多的 JavaScript 框架或者库都是这样做的）。一旦这样就会陷入了新的困局，Vue 是不会维护 window 的这些共享变量的。对于组件来讲，这些变量都存在于组件作用域以外，组件并不会“多管闲事”替我们托管。那我们就不得不手工来接管这些变量的赋值与读取。然而，只要我们知道一些基本的 JS 编程规范或者风格规范都会明白这么一条准则：全局变量是毒瘤，是具有极高副作用的。

这样讲可能还不容易理解，还是按照本书的风格以示例说话。举一个经常遇到的例子：用户对象的共享。由于登录逻辑都放在后台，为了减少前后台的请求数量，当页面加载时我们经常通过服务端页面直接将一个用户对象输出为 JSON 并保存到 window 内，让客户端代码无须频繁地向服务器获取当前用户的对象数据。以 Rails 为例，在用户登入成功后在 Vue 的引导页面中加入下面的代码：

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
  <script>
    // 将服务端的 @user 对象变成浏览器端的 current_user
    window.$data = { current_user :
  </script>
</head>
<body>
  <div id="app">
    <!-- 用于挂载 vue 的元素容器-->
  </div>
</body>
</html>
```

当完成了这一步以后，相当于用 window.\$data 作为服务端对象到客户端的一个输出出口，将变量从服务端“传递”到客户端，接下来就可以在 Vue 组件内将这个 \$data 变成 Vue 的 data：

```
import _ from 'lodash'

export default {
  data () {
    return _.extend({
      current_user: {
        is_auth: false
      }
    },
    window.$data)
  },
  // ...
}
```

这种做法是一种很常用也很实用的技巧。因为确实可以省去不少在 created 钩子中用 \$http 调用服务端 API 的初始化动作。当不使用 vue-router 的时候，或者说 window.\$data 的变量是完全只读的时候，这种方法是没有任何副作用的。

然而，只要将 `window.$data` 内的对象绑定到不同的自定义组件内，一旦要对 `window.$data` 内的变量进行修改，那么你的噩梦就开始了——你会发现所有以对象方式绑定的自定义组件，当对象内的某个属性发生改变时将不会执行自动刷新，所有的计算属性也同时失效！更诡异的是这种情况并不是绝对出现的，当页面元素相对简单的时候一切都显得很正常，一旦页面元素增多，对应的交互操作增多时，这种奇怪的现象就会出现。

例如，对于 `window.$data.current_user.has_new_mail` 这个字段，我们定义一个这样的组件：

```
<template>
  <div>
    <span v-if="user.has_new_mail">你有新邮件</span>
  </div>
</template>
<script>
  export default {
    props: ['user']
  }
</script>
```

并且建立这样的一个父组件来进行绑定：

```
<tempalte>
  <div>
    <email-notifier :user="current_user"></email-notifier>
  </div>
</tempalte>
<script>
import _ from 'lodash'
import EmailNotifier from './EmailNotifier'

export default {
  data () {
    return _.extend({
      current_user: {
        is_auth: false,
        has_new_mail: false
      }
    },
    window.$data)
  },
  components : {
    EmailNotifier
  }
}
```



```
}  
</script>
```

此时, 如果对 `current_user` 进行赋值, `email-notifier` 是会产生任何变化的, 交互式刷新完全失败! 然而这只是将问题最小化至一个组件中, 当有很多组件都在共享这个 `window.$data.current_user` 对象时, 将会非常难以控制它的改变与 DOM 的刷新。

这种类似的问题在我没有使用 `Vuex` 之前开始不断地在项目中出现, 直至我下决心将状态变量交由 `Vuex` 来管理, 这种变量共享问题才得以真正地解决。

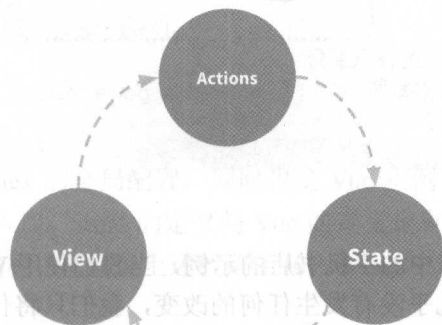
7.1 Vuex 的基本结构

`Vuex` 是一个专为 `Vue.js` 应用程序开发的状态管理模式。它采用集中式存储来管理应用所有组件的状态, 并以相应的规则来保证状态以一种可预测的方式发生变化。`Vuex` 也集成到了 `Vue` 的官方调试工具 `devtools extension` 中, 提供了诸如零配置的 `time-travel` 调试、状态快照导入导出等高级调试功能。

这个状态自管理应用包含以下几个部分:

- `state`——驱动应用的数据源, 也就是各种共享变量;
- `view`——以声明方式将 `state` 映射到视图, 也就是在 `<template>` 模块上引用这些 `state`, 让 `Vuex` 自动处理重绘;
- `actions`——响应在 `view` 上用户输入导致的状态变化。

以下是一个表示“单向数据流”理念的示意图:



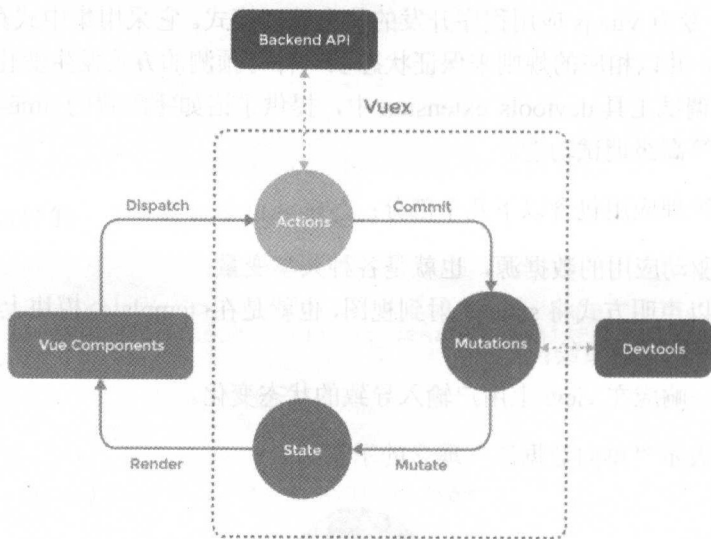
但是, 当应用遇到多个组件共享状态时, 单向数据流的简洁性很容易被破坏:

- 多个视图依赖于同一状态。
- 来自不同视图的行为需要变更为同一状态。

对于问题一，传参的方法对于多层嵌套的组件将会非常烦琐，并且对于兄弟组件间的状态传递无能为力。对于问题二，我们经常会采用父子组件直接引用，或者通过事件来变更和同步状态的多份副本的方式。以上的这些模式非常脆弱，通常会产生无法维护的代码。

因此，为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种模式下，组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为！

另外，通过定义和隔离状态管理中的各种概念并强制遵守一定的规则，代码将会变得更结构化且易维护。



安装 Vuex

```
$ npm i vuex -S
```

示例说明

此处我们将使用第3章中的手机书店的示例，当我们使用 Vuex 对此示例进行改写时，你会发现所有的视图模板几乎没有发生任何的改变，我们只将代码内的“状态”提取到一个可以由 `this.store` 访问的公共状态对象内，其次我们会将所有的 `data` 用计算属性加以取代，当状态发生改变时，Vue 的响应式系统依然按照原有的工作方式自动跟踪状态变化并对视图进行自动重绘，而当我们需要对状态进行改变时则是调用 store 内的 Mutation 进行的。

在 main.js 全局配置文件内引入 Vuex 并进行基本的配置：

```
// main.js
import Vue from 'vue'
import App from './App.vue'
import store from './store'

new Vue({
  el: '#app',
  store,
  render: h => h(App)
})
```

这个配置语法相信你现在已经很熟悉了，正如前文中的 vue-router、vue-resource 和 vee-validate 等库一样，在全局 Vue 实例中引入 store 后，在每个 Vue 组件实例内，可以用 this.\$store 来引用这个 store 对象了。

先建立一个 store 并保存于~/src/store/index.js 文件内，具体内容如下：

```
// store/index.js
import Vue from 'vue'
import Vuex from 'vuex'

// 从环境变量判断当前的运行模式
const debug = process.env.NODE_ENV !== 'production'

// 声明引入此库文件的 Vue 实例使用 Vuex 插件应用状态管理
// 这样可以从 main.js 文件内减少对 vuex 库的依赖
Vue.use(Vuex)

// 导出 store 实例对象
export default new Vuex.Store({
  strict: debug, // 设置运行模式
  plugin: debug ? [createLogger()] : [] // 调试模式则加入日志插件
});
```

简言之，store 就是 Vuex 的全局配置，同时也是 Vue 实例（this）访问 Vuex 获取状态共享服务的公共程序调用入口。Store 的定义与 Vue 的定义是有很多相似之处的，以下就是一个完整的 Store 定义的结构（也就是上述代码中传入 Vuex.Store(options)方法的 options 参数）：

```
export default {
  state: {},
  actions: {}
```

```
  getters: {},  
  mutations: {},  
  modules: {},  
  strict: true,  
  plugin []  
}
```

- **state**——Vuex store 实例的根状态对象，用于定义共享的状态变量，就像 Vue 实例中的 data。
- **getters**——读取器，外部程序通过它获取变量的具体值，或者在取值前做一些计算（可以认为是 store 的计算属性）。
- **actions**——动作，向 store 发出调用通知，执行本地或者远端的某一个操作（可以理解为 store 的 methods）。
- **mutations**——修改器，它只用于修改 state 中定义的状态变量。
- **modules**——模块，向 store 注入其他子模块，可以将其他模块以命名空间的方式引用。
- **strict**——用于设置 Vuex 的运行模式，true 为调试模式，false 为生产模式。
- **plugin**——用于向 Vuex 加入运行期的插件。

由 store 的定义其实就可以看出一些结构上的端倪，store 可以通过 modules 的注册形成树状的实例结构。我们必须要先建立这样一个概念，后面的内容就好理解了。modules 的具体使用在下文会有详细的交待。

7.2 data 的替代者——State 和 Getter

我们来为图书对象建立一个模块，将其保存于~/src/modules/books.js 中。先定义一个 state 对象作为读取图书类数据的容器，简单地说就是将各个 Vue 页面实例内凡是用于保存图书数据的 data 属性一并提取到以下这个文件中，并统一放在 state 变量内，具体做法如下：

```
// ~/src/modules/books.js  
export default {  
  state: {  
    announcements: [],  
    promotions: [],  
    recommended: []  
  }  
}
```

由于 Mutation 是为了改变 state 内的状态而存在的，为了不引起理解上的混淆，在此暂

时先将其放下，后面再对它进行详细的讲解。

我们先从 `Home.vue` 文件开始，将原有的代码重构为使用 `Vuex` 的方式。从原则上来讲，视图模板是不需要做出任何改动的。我们首先要做的是将从 `data` 获取数据改由计算属性去处理。以下是 `Home.vue` 原有的 `data` 定义：

```
export default {
  data () {
    announcements:[],
    promotions:[],
    recommended:[]
  },
  computed : {
    promotionCount () {
      return this.promtions.length
    },
    recommendedCount () {
      return this.recommendedCount.length
    }
  }
  // ... 省略
}
```

接入 `Vuex` 并改由计算属性进行处理：

```
export default {
  computed: {
    announcements () {
      return this.$store.state.announcements
    },
    promotions () {
      return this.$store.state.promotions
    },
    recommended () {
      return this.$store.state.recommended
    },
    promotionCount () {
      return this.$store.state.promtions.length
    },
    recommendedCount () {
      return this.$store.state.recommendedCount.length
    }
  }
}
```


显然，如果每个使用到图书数据的页面都要这样改写定义的话，代码会变得很啰唆。而且 `promotionCount` 和 `recommendedCount` 这两个计算属性如果在多个地方使用的话，就要复制。复制是程序员的大忌，好代码绝对不能出现任何的复制！凡是可以被复制的代码就说明其可以被封装且有重用的需要，因此我们可以采用另一个方法从 `state` 中获取这些变量，这就是 Vuex 中的读取器（Getter）。我们将上述代码统一封装到图书模块内并以 Getter 的方式将它们重新暴露出来：

```
// ~/src/modules/books.js
export default {
  state: {
    announcements: [],
    promotions: [],
    recommended: []
  },
  getters: {
    announcements: state => state.announcements,
    promotions: state => state.promotions,
    recommended: state => state.recommended,
    totalPromotions: state => state.promotions.length,
    totalRecommended: state => state.recommended.length
  }
}
```

然后 `Home.vue` 就可以改写为：

```
export default {
  computed: {
    announcements () {
      return this.$store.getters.announcements
    },
    promotions () {
      return this.$store.getters.promotions
    },
    recommended () {
      return this.$store.getters.recommended
    },
    promotionCount () {
      return this.$store.getters.totalPromotions
    },
    recommendedCount () {
      return this.$store.getters.totalRecommended
    }
  }
}
```

将从状态值直接读取数据转换成从读取器获取数据看起来似乎没有很大的区别，但只要你细想一下 Vue 实例中 data 与 computed 两者之间的关系，你就能理解了——state 相当于 store 实例的 data，而 getters 就相当于 store 实例的 computed。

进一步用 Vuex 提供的帮助方法 mapGetters 简化上述代码：

```
import {mapGetters} from 'vuex'

export default {
  computed: {
    ...mapGetters([
      'announcements',
      'promotions',
      'recommended',
      'promotionCount',
      'recommendedCount'
    ])
  }
}
```

mapGetters 本质上就是动态方法生成器，作用就是生成上面那些将 store.getter 方法映射为 Vue 实例的 computed。

上述代码中采用了 ES6 stage-3 阶段中的对象展开符，如果你学过 Python，它的作用就相当于**object，也就是将一个对象进行解体并展开为多个方法。

这只是为了能让你了解 Getter 存在的意义，才在此例中有意地写出几个毫无意义的 Getter。如果这些 Getter 只是单纯地返回状态值的话，我们可以不定义 Getter，而使用 mapState 帮助方法将所有状态直接映射为计算属性：

```
export default {
  state: {
    announcements:[],
    promotions:[],
    recommended:[]
  },
  getters: {
    totalPromotions: state => state.promotions.length,
    totalRecommended: state => state.recommended.length
  }
}
```

Home.vue 应简化为：

```
import {mapGetters} from 'vuex'
```

```
import _ from 'lodash'

export default {
  computed: {
    ...mapState([
      'announcements',
      'promotions',
      'recommended']),
    ...mapGetters([
      'promotionCount',
      'recommendedCount'
    ])
  }
}
```

7.3 测试 Getter

如果你的 Getter 带有复杂计算，那么测试它们是值得的。Getter 本质上只是一个普通的 JS 函数，并不需要做什么特殊的配置，就当作普通的单元测试来写就可以了。

以下的 Getter 用于从状态变量数组中筛选出指定类别的图书：

```
// getters.js
export const getters = {
  filteredProducts (state, { filterCategory }) {
    return state.products.filter(product => {
      return product.category === filterCategory
    })
  }
}
```

单元测试：

```
// test/unit/store/book-getters.spec.js
import { getters } from '../getters'

describe('getters', () => {
  it('filteredProducts', () => {
    // 准备仿真的状态数据
    const state = {
      products: [
        { id: 1, title: '揭开数据真相：从小白到数据分析达人', category: '大数据' },
        { id: 2, title: '淘宝天猫电商运营与数据化选品完全手册', category: '电商运
```

```
营' },
    { id: 3, title: '大数据架构详解: 从数据获取到深度学习', category: '大数据' }
  ]
}

const filterCategory = '大数据'

// 直接调用 getter
const result = getters.filteredProducts(state, { filterCategory })

// 对结果进行深度断言
expect(result).to.deep.equal([
  { id: 1, title: '揭开数据真相: 从小白到数据分析达人', category: '大数据' },
  { id: 3, title: '大数据架构详解: 从数据获取到深度学习', category: '大数据' }
])
}))
})
```

关于 Promise polyfill 的问题

加入 Vuex 后一旦通过 Karma 启用单元测试，我们会接收到以下的出错信息：

```
[vuex] vuex requires a Promise polyfill in this browser.
```

可以安装一个 Babel 的 polyfill 库来解决这个问题，先安装 babel-polyfill：

```
$ npm i babel-polyfill -D
```

然后在 Karma.conf.js 的 file 设项内加入以下代码：

```
config.set({
  files: [
    '../node_modules/babel-polyfill/dist/polyfill.js',
    './index.js'],
  // ... 省略
})
```

7.4 Action——操作的执行者

此时你可能会产生这样的疑惑，那应该在哪里使用 `vue-resource` 从服务器读取数据并分别写到 `state` 的各个属性中呢？首先继续强调的一点是，一旦引入 Vuex 后，我们的 Vue 实例不能直接修改 `$store.state` 内的任意内容，要修改状态就要通过 `Mutation` 来修改。在本

例中，在修改这些状态数据之前，我们需要执行与服务器之间的通信。那么在使用 Mutation 之前，我们得先做一些 Action 来完成这个需要。

用上文的对应法来理解动作（Action）的话，我们可以将它看作 \$store 的 methods。接下来就开始定义从服务器中读取数据的动作（Actions）：

```
// ~/src/modules/books.js
import Vue from 'vue'

export default {
  state: {
    announcements:[],
    promotions:[],
    recommended:[]
  },
  getters: {
    totalPromotions: state => state.promotions.length,
    totalRecommended: state => state.recommended.length
  },
  actions: {
    getStarted (context) {
      Vue.http.get('/api/get-start', (res)=>{
        context.commit('startedDataReceived', res.body)
      })
    }
  }
}
```

开始调用 vue-resource 时，你可能会变得无所适从。在前文中我们都是通过 Vue 实例的上下文 this.\$http 来获取 vue-resource 的对象实例，可现在 store 实例中的 this 并不是一个 Vue 实例，所以我们得重新导入 Vue 对象的引用，直接从 Vue.http 里获取 vue-resource 实例：

```
Vue.http.get('/api/get-start', (res) => {
  // ...
})
```

Action 是不能直接修改 state 中的状态的，每个 Action 定义的第一个参数必然是一个与当前 store 实例结构相同的 context 对象，这个对象具有以下属性：

- state——等同于 store.state，若在模块中则为局部状态。
- rootState——等同于 store.state，只存在于模块中。

- `commit`——等同于 `store.commit`，用于提交一个 `mutation`。
- `dispatch`——等同于 `store.dispatch`，用于调用其他 `action`。
- `getters`——等同于 `store.getters`，获取 `store` 中的 `getters`。

我们只调用 `context.commit` 提交了一个 `startedDataReceived` 并传入 AJAX 调用返回的 JSON 对象 `res.body`，这里是否有点触发事件的意味？事实正是如此！我们可以将 `commit` 方法看作 `Action` 的一个执行终点，它只是负责通知 Vuex 执行完一个动作或者处理完成，这个动作产生的结果就是输入的参数，交由名为 `startedDataReceived` 的 `Mutation` 继续进行下一步的处理。

如果实际面对的业务很复杂，那么可能一个 `Action` 内的操作就不止是像上述代码中单纯地进行远程方法调用了，有可能在这个 `Action` 中读取其他 `state` 对象作为输入参数，也可能需要执行另外一个 `Action`。简言之，`Action` 除了像 Vue 实例中的 `methods`，更像是 MVC 模式中的 `Controller`，唯一的区别只是 `Action` 可以执行任何的异步处理，并且它只对状态进行读取而不做出任何修改。

7.5 测试 Action

测试 `Action` 就有点棘手了，因为它们可能依赖外部 API。当测试 `Action` 的时候，我们通常需要做某种程度的 `mocking`（模拟）——例如，我们可以把 API 调用抽象封装到一个服务类，然后在测试中模拟这个服务。为了简单模拟依赖，我们可以使用 `webpack` 和 `inject-loader` 类打包我们的测试文件。

测试异步 `Action` 的例子：

```
// actions.js
import shop from '../api/shop'

export const getAllProducts = ({ dispatch }) => {
  dispatch('REQUEST_PRODUCTS')
  shop.getProducts(products => {
    dispatch('RECEIVE_PRODUCTS', products)
  })
}

// actions.spec.js

// use require syntax for inline loaders.
// with inject-loader, this returns a module factory
```

```
// that allows us to inject mocked dependencies.
import { expect } from 'chai'
const actionsInjector = require('inject!./actions')

// create the module with our mocks
const actions = actionsInjector({
  '../api/shop': {
    getProducts (cb) {
      setTimeout(() => {
        cb([ /* mocked response */ ])
      }, 100)
    }
  }
})

// helper for testing action with expected mutations
const testAction = (action, args, state, expectedMutations, done) => {
  let count = 0

  // mock commit
  const commit = (type, payload) => {
    const mutation = expectedMutations[count]
    expect(mutation.type).to.equal(type)
    if (payload) {
      expect(mutation.payload).to.deep.equal(payload)
    }
    count++
    if (count >= expectedMutations.length) {
      done()
    }
  }

  // call the action with mocked store and arguments
  action({ commit, state }, ...args)

  // check if no mutations should have been dispatched
  if (expectedMutations.length === 0) {
    expect(count).to.equal(0)
    done()
  }
}

describe('actions', () => {
  it('getAllProducts', done => {
```

```

testAction(actions.getAllProducts, [], {}, [
  { type: 'REQUEST_PRODUCTS' },
  { type: 'RECEIVE_PRODUCTS', payload: { /* mocked response */ } }
], done)
})
})

```

7.6 只用 Mutation 修改状态

当我们从远程数据库读取完数据后就要通知 store 调用一个 Mutation 来修改状态，也就是上文中的 `startedDataReceived`，在 Action 中的写法有点像在 Vue 实例内调用 `$emit` 触发事件。确实从理解上来说，Mutation 就像是 store 内的专属事件，它只能由 store 本身进行回调，当然也可以通过 `store.commit` 方法直接触发一个 Mutation。

加入 mutations:

```

// ~/src/modules/books.js
import Vue from 'vue'

export default {
  state: {
    announcements: [],
    promotions: [],
    recommended: []
  },
  getters: {
    totalPromotions: state => state.promotions.length,
    totalRecommended: state => state.recommended.length
  },
  actions: {
    getStarted (context) {
      Vue.http.get('/api/get-start', (res) => {
        context.commit('startedDataReceived', res.body)
      })
    }
  },
  mutations: {
    startedDataReceived (state, {started_data}) {
      state.announcements = started_data.announcements
      state.promotions = started_data.promotions
      state.recommended = started_data.recommended
    }
  }
}

```

```

    }
  }
}

```

因此，在 actions 的下方我定义了一个名为 startedDataReceived 的 Mutation 回调处理方法，使之与 getStarted Action 内提交的 startedDataReceived 对应。最后，上述代码中有一个地方是重复性非常强的，就是 startedDataReceived 的定义与引用，它们的出现频率将是 mutation 定义的两倍以上，为了避免这种可怕而低级的重复性的出现，我们可以用 ES6 的常量定义来修改一下上述的代码，直接将 Mutation 作为一种事件常量来使用：

```

//~/src/store/mutation-types.js
export const STARTED_DATA_RECEIVED = 'startedDataReceived'

```

改用常量定义：

```

// ~/src/modules/books.js
import Vue from 'vue'
import {STARTED_DATA_RECEIVED} from '../mutation-types'

export default {
  state: {
    // ...省略
  },
  getters: {
    // ...省略
  },
  actions: {
    getStarted (context) {
      Vue.http.get('/api/get-start', (res)=>{
        context.commit(STARTED_DATA_RECEIVED,res.body)
      })
    },
  },
  mutations: {
    [STARTED_DATA_RECEIVED] (state,{started_data}) {
      // ...省略
    }
  }
}
};

```

最后，回到 Home.vue 中来调用 Action，以初始化共享状态中的数据。Action 也是不能直接调用的，它只能通过 Vuex 的 dispatch 方法向 Vuex 发出调用通知，由 Vuex 来找到这个指定的 Action 并执行。


```
import {mapGetters} from 'vuex'

export default {
  computed: {
    ...mapGetters([
      'announcements',
      'promotions',
      'recommended',
      'promotionCount',
      'recommendedCount'
    ])
  },
  created () {
    this.$store.dispatch('getStarted')
  }
}
```

如前文所提及的，Action 相当于 Vue 的 methods。同样地，Vuex 也提供了将 Actions 映射为 methods 的帮助方法 mapActions，这样我们就不用直接调用 dispatch，而是可以对象化地进行操作了：

```
import {mapGetters, mapActions} from 'vuex'
import _ from 'lodash'

export default {
  computed: {
    ...mapState([
      'announcements',
      'promotions',
      'recommended'
    ]),
    ...mapGetters([
      'promotionCount',
      'recommendedCount'
    ])
  },
  methods: {
    ...mapActions(['getStarted'])
  },
  created () {
    this.getStarted()
  }
}
```


Vue2 中的属性绑定都是单向的，也就是说，绑定到组件属性上的变量必须是只读的、不可改变的（Immutable），只有这样的变量才没有副作用。

7.7 测试 Mutations

Mutations 很好测试，因为它们就是仅依赖参数本身的方法。有一个技巧就是，如果用 ES2015 的模块化并且把 Mutation 放到 store.js 文件里，除了默认的 export，你还可以把这个 Mutation 以一个命名参数来“export”。

```
const state = { ... }

// 命名参数 export mutation
export const mutations = { ... }

export default new Vuex.Store({
  state,
  mutations
});
```

使用 Mocha + Chai 测试一个 Mutation 的例子：

```
// mutations.js
export const mutations = {
  increment: state => state.count++
}

// mutations.spec.js
import { mutations } from './store'

// destructure assign mutations
const { increment } = mutations

describe('mutations', () => {
  it('INCREMENT', () => {
    // mock state
    const state = { count: 0 }
    // apply mutation
    increment(state)
    // assert result
    expect(state.count).to.equal(1)
  })
})
```

7.8 子状态和模块

前文中我们展示了如何在一个\$store 实例内直接定义购物车的状态，我们称其为“单一状态树”。此时 `getter`、`action` 和 `mutation` 都是围绕购物车的行为及状态修改而定义的。从这个示例就可以全面地了解操作一个“模型 (Model)”所需要的最小的 Vuex 结构是怎样的，下面就全面回顾一下 `store` 的全部代码定义：

```
//~/src/modules/cart.js
import Vue from 'vue'
import * as types from '../mutation-types'
import * as apis from '../apis'

export default {
  state: {
    all: []
  },
  getters: {
    cartItems () {
      return state.all
    }
  },
  actions: {
    addToCart (context, {id}) {
      Vue.http
        .post(apis.ADD_TO_CART, {id:id})
        .then(res => {
          context.commit(types.ADD_TO_CART, res.body)
        })
    },
    getCartItems (context) {
      Vue.http
        .get(apis.GET_CART_ITEMS)
        .then(res => {
          context.commit(types.SET_CART_ITEMS, res.body.data)
        })
    },
    clear (context) {
      Vue.http
        .post(apis.CLEAR_CART)
        .then(res => {
          context.commit(type.SET_CART_ITEMS, [])
        })
    }
  }
}
```

```
    },  
    mutations: {  
      [types.SET_CART_ITEMS] (state,{items}) {  
        state.all = items  
      },  
      [types.ADD_TO_CART] (state,{cartItem}) {  
        state.all.push(cartItem)  
      }  
    }  
  }  
}
```

真正的项目所需要处理的模型远远不止一个，一个商城除了购物车，还需要定义用来存取商品信息商品（图书）模型；需要用来记录商品 SKU 的库存模型；需要记录交易信息的订单模型，等等。如果向现在的 store 中加入对图书商品和订单的状态支持，首先就要向 state 对象加入两个不同的 all 状态，用于获取所有的图书与订单。显然这样就会导致命名冲突，那么就不得不将这三个状态用以下的方式来命名：

```
export default {  
  state: {  
    allCartItems: [],  
    allBooks: [],  
    allOrders: []  
  },  
  // ... 省略  
}
```

而它们的 action 和 mutation 也会面对类似的问题。从语言级别上而言，这是没有命名空间所导致的大量不得已的含混性重复，随着项目变得越来越大，这种含混性的重复也将以倍数级别增长，store 会变得非常臃肿且难以维护。幸好 Vuex 提供了“模块”的定义，通过模块，我们可以将各种在功能使用上具有独立性的模型状态，将各自的模型状态作为子状态挂入至 store 根状态内，以模块名称作为命名空间来使用。这样做的话，操作和变更就被划分到各自模块内，由模块提供的命名空间进行引用，从而使 store 的结构与代码组织变得清晰和容易理解。

简言之，模块可看作一个独立处理某个业务对象的 store，而 store.js 文件就作为一个文件入口，是所有业务的“根”，就像 main.js 文件那样用作一种全局性的配置，在运行期组合出真正完整的 store 对象。要达到这样的效果，就需要对 store 的工程文件结构进行重整。Vue 官方推荐了以下的文件结构的组织方式及使用原则：

- (1) 应用级的状态集中在 store 中。
- (2) 修改状态的唯一方式就是通过提交 mutation 来实现的，它是同步的事务。

(3) 异步逻辑应该封装在 `action` 中，并且可以组合 `action`。

只要遵循这些规则，可以任意设计项目结构。如果 `store` 文件非常大，直接开始分割 `action`、`mutation` 和 `getter` 到多个文件。

对于复杂的应用，我们可能需要使用模块化。下面是一个项目结构：

```

├─ index.html
├─ App.vue
├─ main.js
├─ components
├─ ...
└─ store
  ├─ index.js           # 配置 store 的配置文件
  ├─ actions.js        # 公共根 store 的 actions
  ├─ mutations.js      # 公共根 store 的 mutations
  ├─ mutation-types.js # 公共 mutations 名称定义
  └─ modules           # 模块文件夹
    ├─ cart.js         # 模块定义
    └─ products.js     # ...

```

用以上这种组织型重新改写 `store` 和各自的模块，首先入口文件 `store.js` 的定义将变成以下的方式：

```

import Vue from 'vue'
import Vuex from 'vuex'
import createLogger from 'vuex/dist/logger'
import * as actions from './actions'
import * as getters from './getters'
import cart from './modules/cart'
import products from './modules/products'

const debug = process.env.NODE_ENV !== 'production'

Vue.use(Vuex)

export default new Vuex.Store({
  modules: {
    cart,
    products
  },
  actions: actions,
  getters: getters,
  strict: debug, // 设置运行模式
  plugin: debug ? [createLogger()] : [] // 调试模式则加入日志插件
});

```


~/src/store/modules/cart.js 就会变为:

```
import Vue from 'vue'
import * as apis from '../apis'
import * as types from '../mutation-types'

const state = {
  all: []
}

const mutations = {
  [types.SET_CART_ITEMS] (state, cart) {
    state.all = cart.items
  }
}

const actions = {
  getItems (context) {
    Vue.http.get(apis.GET_CART_ITEMS)
      .then(res => {
        context.commit(types.SET_CART_ITEMS, res.body.data)
      })
  }
}

export default {
  state,
  mutations,
  actions
}
```

~/src/modules/products.js 模板的定义如下:

```
import Vue from 'vue'
import * as apis from '../apis'
import * as types from '../mutation-types'

const state = {
  announcement: {},
  top: [],
  promotions: [],
  recommended: []
}
```



```

const mutations = {
  [types.SET_BOOK] (state, book) {
    state.book = book
  },

  [types.STARTED_DATA_RECEIVED] (state, {top, announcement, promotions, recommended}) {
    state.top = top
    state.announcement = announcement
    state.promotions = promotions
    state.recommended = recommended
  }
}

const actions = {
  getBook (context, id) {
    Vue.http.get(apis.GET_BOOK, {id}).then(res => {
      context.commit(types.SET_BOOK, res.body)
    })
  },

  getStarted (context) {
    Vue.http.get(apis.GET_STARTED).then(res => {
      context.commit(types.STARTED_DATA_RECEIVED, res.body)
    })
  }
}

export default {
  state,
  actions,
  mutations
}

```

当文件结构重新组织和整理后，对 `cart` 和 `product` 的引用方式也要稍加变化，只需要在对应的 `state` 内先加入与模块同名的对象名称作为命名空间就可以直接引用子模块内的对象：

- 引用子状态——`this.$store.state.cart.all`。
- 引用子读取器——`this.$store.getters.cart.allItems`；
- 引用子动作——`this.$store.actions.cart.getAll`；

- 引用子变更——`this.$store.mutations.cart[types.SET_BOOK]`。

那么子读取器、子动作和子变更也可以这样引用吗？

答案是否定的！事实上，只有子状态可以通过命名空间引用！我认为这是 Vuex 设计上的一大遗憾！如果我们用调试器观察 `$store` 变量内的内容，就会发现这一真相。`getters`、`actions` 和 `mutations` 并没有任何与模块命名相同的对象命名空间存在，也就是说，如果通过 `this.$store.getters.cart.allItems` 引用，你将会得到一个 `undefined` 的值！`allItems` 仍然被放置于最顶层的 `getters` 变量内，这一点你需要清楚地了解，因为这样就会使 Action 和 Getters 不得不使用命名前缀来进行区分。

动态注册模块

除了在入口文件 `store.js` 中对模块进行注册，在某些应用场景中可能需要向现有的 `store` 内注册新的自定义模块，在这种情况下可以使用 `registerModule` 方法，以编程方式动态地向 `store` 实例注册新的模块：

```
store.registerModule('books', {  
  // ...  
})
```

模块动态注册功能可以让其他 Vue 插件向已有的 `store` 附加新模块，以此来分割 Vuex 的状态管理。例如，`vuex-router-sync` 插件可以集成 `vue-router` 与 Vuex，管理动态模块的路由状态。

也可以使用 `store.unregisterModule(moduleName)` 动态地卸载模块。注意，不能使用此方法卸载静态模块（在创建 `store` 时声明的模块）。

7.9 用服务分离外部操作

当我们开始使用 Vuex 分离原有大量被掺杂在组件当中的各种状态管理之后，会慢慢地觉得 Vuex 并不是那么难以理解，也会体验到它确实让 Vue 回归了本原：一个制作界面的框架。在进行 Vuex 编程的过程中，不知道你是否会发现 Actions 内有大量相同的或者相似的 `Vue.http` 的远程方式调用。它们显得是如此的类似、不雅，甚至是让人感到臃肿。

在实际的 Vue 项目开发过程中，RESTful API 调用可以说是无处不在，在 Vue 的生态圈内除了前文中推荐使用的 `vue-resource`，还有其他一些功能类似的 AJAX 包，可以到 <https://github.com/vuejs/awesome-vue> 上找到它们。另外这个 Repository 集中了所有 Vue 中

各类出色的包，是一个非常值得收藏并参与的资源。

言归正传，对于这些像面条一样揉搓在我们代码中的 AJAX 调用，我们可以引用 Angular 中的服务概念来进行重构，虽然 Angular 的服务是通过反转注入的模式实现的，但模式的过度使用有时并不会减少开发代价，而优秀的思路则不然。因此，我们可以用 Vue 的方式构建一个服务引用结构来重新将所有的远程方法调用进行有效的重构。

借用 Vuex 的文件结构组织方式，建立一个独立的 services 文件夹，独立存放所有的 API 调用相关的文件：

```

├── index.html
├── App.vue
├── main.js
├── components
└── services
    ├── index.js      # 配置 service 的入口文件
    ├── apis.js       # API 地址引用
    ├── cart.js       # 购物车相关的 RESTful API
    ├── product.js    # 图书（产品）相关的 RESTful API
    └── ...

```

然后构造一个 service (index.js) 入口对象：

```

import _ from 'lodash'
import Vue from 'vue'
import VueResource from 'vue-resource'
import product from './product.js'
import cart from './cart.js'

```

```
Vue.use(VueResource)
```

```

export default {
  product,
  cart
}

```

product.js 的内容如下：

```

import Vue from 'vue'
import * as apis from './apis'

export default {
  get: (id) => Vue.http.get(apis.GET_BOOK, {id})
  getStarted: () => Vue.http.get(apis.GET_START)
}

```

cart.js 的代码如下:

```
import Vue from 'vue'
import * as apis from './apis'

export default {
  getAllItems: (id) => Vue.http.get(apis.GET_CARTITEMS)
}
```

~/src/store/modules/cart.js 就会变为:

```
import * as types from '../mutation-types'
import services from '../services'

const state = {
  // ... 省略
}

const mutations = {
  // ... 省略
}

const actions = {
  getItems (context) {
    services.cart.getAllItems.then(res => {
      context.commit(types.SET_CART_ITEMS, res.body.data)
    })
  }
}

export default {
  state,
  mutations,
  actions
}
```

这里采用的是面向对象的一点小技巧, 以命名空间的方式有效地重新组织所有 API 服务, 旨在重新合理规划我们的代码, 以适应不断膨胀的项目。

附录 A Chai 断言参考

not

对之后的断言取反：

```
expect(foo).to.not.equal('bar')
expect(goodFn).to.not.throw(Error)
expect({ foo: 'baz' }).to.have.property('foo').and.not.equal('bar')
```

deep

设置 `deep` 标记，然后使用 `equal` 和 `property` 断言。该标记可以让其后的断言不是比较对象本身，而是递归比较对象的键值对：

```
expect(foo).to.deep.equal({ bar: 'baz' })
expect({ foo: { bar: { baz: 'quux' } } })
  .to.have.deep.property('foo.bar.baz', 'quux')
```

`deep.property` 中的特殊符号可以使用双反斜杠进行转义（第一个反斜杠是在字符串参数中对第二个反斜杠进行转义，第二个反斜杠用于在 `property` 中进行转义）：

```
var deepCss = { '.link': { '[target]': 42 } }
expect(deepCss).to.have.deep.property('\\.link\\.\\[target\\]', 42)
```

any

在 `keys` 断言之前使用 `any` 标记（与 `all` 相反）：

```
expect(foo).to.have.any.keys('bar', 'baz')
```

all

在 `keys` 断言之前使用 `all` 标记（与 `any` 相反）：

```
expect(foo).to.have.all.keys('bar', 'baz')
```

a(type) / an(type)

- `type`: String, 被测试的值的类型。

`a` 和 `an` 断言即可作为语言链又可作为断言使用：


```
// 类型断言
expect('test').to.be.a('string');
expect({ foo: 'bar' }).to.be.an('object');
expect(null).to.be.a('null');
expect(undefined).to.be.an('undefined');
expect(new Error).to.be.an('error');
expect(new Promise).to.be.a('promise');
expect(new Float32Array()).to.be.a('float32array');
expect(Symbol()).to.be.a('symbol');

// es6 overrides
expect({ [Symbol.toStringTag]: ()=>'foo' }).to.be.a('foo');

// language chain
expect(foo).to.be.an.instanceof(Foo);
** include(value) / contains(value) **
```

- value: Object / String / Number.

`include()`和 `contains()`既可作为属性类断言前缀语言链又可作为作为判断数组、字符串是否包含某值的断言使用。当作为语言链使用时，常用于 `key()` 断言之前：

```
expect([1, 2, 3]).to.include(2)
expect('foobar').to.include('bar')
expect({ foo: 'bar', hello: 'universe' }).to.include.keys('foo')
```

ok

断言目标为真值：

```
expect('everything').to.be.ok
expect(1).to.be.ok
expect(false).to.not.be.ok
expect(null).to.not.be.ok
```

true

断言目标为 `true`，注意，这里与 `ok` 的区别是不进行类型转换，只有为 `true` 才能通过断言：

```
expect(true).to.be.true
expect(1).to.not.be.true
```

false

断言目标为 `false`：

```
expect(false).to.be.false
```

```
expect(0).to.not.be.false
```

null

断言目标为 null:

```
expect(null).to.be.null  
expect(undefined).to.not.be.null
```

undefined

断言目标为 undefined:

```
expect(undefine).to.be.undefined  
expect(null).to.not.be.undefined
```

NaN

断言目标为非数字 NaN:

```
expect('foo').to.be.null  
expect(4).to.not.be.null
```

exist

断言目标存在, 既非 null 也非 undefined:

```
var foo = 'hi',  
    bar = null,  
    baz  
  
expect(foo).to.exist  
expect(bar).to.not.exist  
expect(baz).to.not.exist
```

empty

断言目标的长度为 0。对于数组和字符串, 它检查 length 属性, 对于对象, 它检查可枚举属性的数量:

```
expect([]).to.be.empty  
expect('').to.be.empty  
expect({}).to.be.empty
```

arguments

断言目标是一个参数对象 arguments:

```
function test () {  
  expect(arguments).to.be.arguments
```

```
}
```

equal(value)

- value: Mixed

断言目标严格等于(===)value。另外,如果设置了 deep 标记,则断言目标深度等于 value:

```
expect('hello').to.equal('hello')
expect(42).to.equal(42)
expect(1).to.not.equal(true)
expect({ foo: 'bar' }).to.not.equal({ foo: 'bar' })
expect({ foo: 'bar' }).to.deep.equal({ foo: 'bar' })
```

eql(value)

- value: Mixed

断言目标深度等于 value, 相当于 deep.equal(value)的简写:

```
expect({ foo: 'bar' }).to.eql({ foo: 'bar' })
expect([1, 2, 3]).to.eql([1, 2, 3])
```

above(value)

- value: Number

断言目标大于(超过) value:

```
expect(10).to.be.above(5)
```

也可接在 length 后来断言一个最小的长度。相比直接提供长度的好处是提供了更详细的错误消息:

```
expect('foo').to.have.length.above(2)
expect([1, 2, 3]).to.have.length.above(2)
```

least(value)

- value: Number

断言目标不小于(大于或等于) value:

```
expect(10).to.be.at.least(10)
```

也可接在 length 后来断言一个最小的长度。相比直接提供长度的好处是提供了更详细的错误消息:

```
expect('foo').to.have.length.of.at.least(3)
expect([1, 2, 3]).to.have.length.of.at.least(3)
```

below(value)

- value: Number

断言目标小于 value:

```
expect(5).to.be.below(10)
```

也可接在 `length` 后来断言一个最大的长度。相比直接提供长度的好处是提供了更详细的错误消息:

```
expect('foo').to.have.length.below(4)
expect([1, 2, 3]).to.have.length.below(4)
```

most(value)

- value: String

断言目标不大于（小于或等于）value:

```
expect(5).to.be.at.most(5)
```

也可接在 `length` 后来断言一个最大的长度。相比直接提供长度的好处是提供了更详细的错误消息:

```
expect('foo').to.have.length.of.at.most(4)
expect([1, 2, 3]).to.have.length.of.at.most(3)
```

within(start, finish)

- start: Number, 下限;
- finish: Number, 上限。

断言目标在某个区间内:

```
expect(7).to.be.within(5,10)
```

也可接在 `length` 后来断言一个长度区间。相比直接提供长度的好处是提供了更详细的错误消息:

```
expect('foo').to.have.length.within(2, 4)
expect([1, 2, 3]).to.have.length.within(2, 4)
```

instanceof(constructor)

- constructor: Constructor, 构造函数。

断言目标是构造函数 `constructor` 的一个实例:

```
var Tea = function (name) { this.name = name },
    Chai = new Tea('chai')
```

```
expect(Chai).to.be.an.instanceof(Tea)
expect([1, 2, 3]).to.be.an.instanceof(Array)
```

property(name, [value])

- name: String, 属性名;
- value: Mixed, 可选, 属性值。

断言目标是否拥有名为 name 的属性, 如果提供了 value 则该属性值还需要严格等于 (===) value。如果设置了 deep 标记, 则可以使用点 “.” 和中括号 “[]” 来指向对象和数组中的深层属性:

```
// 简单引用
var obj = { foo: 'bar' }
expect(obj).to.have.property('foo')
expect(obj).to.have.property('foo', 'bar')

// 深层引用
var deepObj = {
  green: { tea: 'matcha' },
  teas: [ 'Chai', 'matcha', { tea: 'konacha' } ]
}

expect(deepObj).to.have.deep.property('green.tea', 'matcha')
expect(deepObj).to.have.deep.property('teas[1]', 'matcha')
expect(deepObj).to.have.deep.property('teas[2].tea', 'konacha')
```

如果目标是一个数组, 还可以直接使用一个或多个数组下标作为 name 来在嵌套数组中断言 deep.property:

```
var arr = [
  [ 'chai', 'matcha', 'konacha' ],
  [ { tea: 'chai' },
    { tea: 'matcha' },
    { tea: 'konacha' }
  ]
]

expect(arr).to.have.deep.property('[0][1]', 'matcha')
expect(arr).to.have.deep.property('[1][2].tea', 'konacha')
```

此外, property 把断言的主语 (subject) 从原来的对象变为当前属性的值, 使得可以在其后进一步衔接其他链式断言 (来针对这个属性值进行测试):

```
expect(obj).to.have.property('foo')
```



```

    .that.is.a('string')
expect(deepObj).to.have.property('green')
    .that.is.an('object')
    .that.deep.equals({ tea: 'matcha' })
expect(deepObj).to.have.property('teas')
    .that.is.an('array')
    .with.deep.property('[2]')
    .that.deep.equals({ tea: 'konacha' })

```

只有当设置了 `deep` 标记的时候，在 `property()` name 中的点 (.) 和中括号 ([]) 才必须使用双反斜杠进行转义（为什么是双反斜杠，在前文有提及），当没有设置 `deep` 标记的时候，是不能进行转义的。

```

// 简单指向
var css = { '.link[target]': 42 }
expect(css).to.have.property('.link[target]', 42)

// 深度指向
var deepCss = { 'link': { '[target]': 42 } }
expect(deepCss).to.have.deep.property('\.link\[.target]', 42)

```

ownProperty(name)

- name: String, 属性名。

断言目标拥有名为 name 的自有属性：

```
expect('test').to.have.ownProperty('length')
```

ownPropertyDescriptor(name[, descriptor])

- name: String, 属性名；
- descriptor: Object, 描述对象，可选。

断言目标的某个自有属性存在描述符对象，如果给定了 `descriptor` 描述符对象，则该属性的描述符对象必须与其相匹配：

```

expect('test').to.have.ownPropertyDescriptor('length')
expect('test').to.have.ownPropertyDescriptor('length', {
  enumerable: false,
  configurable: false,
  writable: false,
  value: 4
})

expect('test').not.to.have.ownPropertyDescriptor('length', {
  enumerable: false,

```

```

    configurable: false,
    writeable: false,
    value: 3
  })

```

// 将断言的主语改为了属性描述符对象

```

expect('test').to.haveOwnPropertyDescriptor('length')
  .to.have.property('enumerable', false)

```

```

expect('test').to.haveOwnPropertyDescriptor('length')
  .to.have.keys('value')

```

length

设置.have.length 标记作为比较 length 属性值的前缀:

```

expect('foo').to.have.length.above(2)
expect([1, 2, 3]).to.have.length.within(2, 4)

```

lengthOf(value)

-value: Number.

断言目标的 length 属性为期望的值:

```

expect([1, 2, 3]).to.have.lengthOf(3)
expect('foobar').to.have.lengthOf(6)

```

match(regex)

- regex: RegExp, 正则表达式。

断言目标匹配到一个正则表达式:

```

expect('foobar').to.match(/^foo/)

```

string(string)

- string: String, 字符串。

断言目标字符串包含另一个字符串:

```

expect('foobar').to.have.string('bar')

```

keys(key1, [key2], [...])

- key: String | Array | Object 属性名。

断言目标包含传入的属性名。与 any、all、contains 或者 have 前缀结合使用会影响测试结果:

- 当与 `any` 结合使用时，无论是使用 `have` 还是使用 `contains` 前缀，目标必须至少存在一个传入的属性名才能通过测试。注意，`any` 或者 `all` 应当至少使用一个，否则默认为 `all`。
- 当结合 `all` 和 `contains` 使用时，目标对象必须至少拥有全部传入的属性名，但是它也可以拥有其他属性名。
- 当结合 `all` 和 `have` 使用时，目标对象必须且仅能拥有全部传入的属性名：

```
// 结合 any 使用
expect({ foo: 1, bar: 2, baz: 3 }).to.have.any.keys('foo', 'bar')
expect({ foo: 1, bar: 2, baz: 3 }).to.contains.any.keys('foo', 'bar')

// 结合 all 使用
expect({ foo: 1, bar: 2, baz: 3 }).to.have.all.keys('foo', 'bar', 'baz')
expect({ foo: 1, bar: 2, baz: 3 }).to.contains.all.keys('foo', 'bar')

// 传入 string
expect({ foo: 1, bar: 2, baz: 3 }).to.have.any.keys('foo')
// 传入 Array
expect({ foo: 1, bar: 2, baz: 3 }).to.have.all.keys(['foo', 'bar', 'baz'])
// 传入 Object
expect({ foo: 1, bar: 2, baz: 3 }).to.have.any.keys({ bar: 2, foo: 1 })
```

throw(constructor)

- `constructor`: `ErrorConstructor` | `String` | `RegExp`。

断言目标函数会抛出一个指定错误或错误类型（使用 `instanceOf` 计算），也可使用正则表达式或者字符串来检测错误消息：

```
var err = new ReferenceError('this is a bad function')
var fn = function () { throw err }

expect(fn).to.throw(ReferenceError)
expect(fn).to.throw(Error)
expect(fn).to.throw(/bad function/)
expect(fn).to.not.throw('good function')
expect(fn).to.throw(ReferenceError, /bad function/)
expect(fn).to.throw(err)
```

注意，当一个抛错断言被否定了（前面有 `not`），那么它会从 `Error` 构造函数开始依次检查各个可能传入的参数。检查一个只是消息类型不匹配但是已知的错误，合理的方式是先断言该错误存在，然后使用 `and` 后断言错误消息不匹配：

```
expect(fn).to.throw(ReferenceError)
    .and.not.throw(/good function/)
```

respondTo(method)

- method: String。

断言目标类或对象会响应一个方法（存在这个方法）：

```
Klass.prototype.bar = function () {}
expect(Klass).to.respondTo('bar')
expect(obj).to.respondTo('bar')
```

如果需要检查一个构造函数是否会响应一个静态方法（挂载在构造函数本身的方法），请查看 `itself` 标记：

```
Klass.baz = function () {}
expect(Klass).itself.to.respondTo('baz')
```

itself

设置 `itself` 标记，然后使用 `respondTo` 断言：

```
function Foo () {}
Foo.bar = function () {}
Foo.prototype.baz = function () {}

expect(Foo).itself.to.respondTo('bar')
expect(Foo).itself.not.to.respond('baz')
```

satisfy(method)

- method: Function，测试器，接受一个参数表示目标值，返回一个布尔值。

断言目标值能够让给定的测试器返回真值：

```
expect(1).to.satisfy(function (num) { return num > 0 })
```

closeTo(expected, delta)

- expect: Numbre，期望值；
- delta: Numbre，范围半径。

断言目标数字等于 `expected`，或在期望值的 \pm -delta 范围内：

```
expect(1.5).to.be.closeTo(1, 0.5)
```

members(set)

- set: Array

断言目标是 `set` 的超集，或前者有后者所有严格相等 (`===`) 的成员。另外，如果设置了 `deep` 标记，则成员进行深度比较（`include/contains` 只能接受单个值，但它们的主语除了

是数组，还可以判断字符串；`members` 则将它们的能力扩展为能够接受一个数组，但主语只能是数组）：

```
expect([1, 2, 3]).to.include.members([3, 2])
expect([1, 2, 3]).to.not.include.members([3, 2, 8])

expect([4, 2]).to.have.members([2, 4])
expect([5, 2]).to.not.have.members([5, 2, 1])
expect([ { id: 1 } ]).to.deep.include.members([ { id: 1 } ])
```

oneOf(list)

- `list`: Array。

断言目标值出现在 `list` 数组的某个顶层位置（直接子元素，严格相等）：

```
expect('a').to.be.oneOf(['a', 'b', 'c'])
expect(9).to.not.be.oneOf(['z'])

// 严格相等，所以对象类的值必须为同一个引用才能被判定为相等
var three = [3]
expect([3]).to.not.be.oneOf([1, 2, [3]])
expect(three).to.not.be.oneOf([1, 2, [3]])
expect(three).to.be.oneOf([1, 2, three])
```

change(object, property)

- `object`: Object, 对象；
- `property`: String, 属性名。

断言目标方法会改变指定对象的指定属性：

```
var obj = { val: 10 }
var fn = function () { obj.val += 3 }
var noChangeFn = function () { return 'bar' + 'baz' }

expect(fn).to.change(obj, 'val')
```

increase(object, property)

- `object`: Object, 对象；
- `property`: String, 属性名。

断言目标方法会增加指定对象的属性：

```
var obj = { val: 10 }
var fn = function () { obj.val = 15 }
expect(fn).to.increase(obj, val)
```


decrease(object, property)

- object: Object, 对象;
- property: String, 属性名。

断言目标方法会减少指定对象的属性:

```
var obj = { val: 10 }  
var fn = function () { obj.val = 5 }  
expect(fn).to.decrease(obj, val)
```

extensible

断言目标对象是可扩展的 (可以添加新的属性):

```
var nonExtensibleObject = Object.preventExtensions({})  
var sealedObject = Object.seal({})  
var frozenObject = Object.freeze({})  
  
expect({}).to.be.extensible  
expect(nonExtensibleObject).to.not.be.extensible  
expect(sealedObject).to.not.be.extensible  
expect(frozenObject).to.not.be.extensible
```

sealed

断言目标对象是封闭的 (无法添加新的属性并且存在的属性不能被删除但可以被修改):

```
var sealedObject = Object.seal({})  
var frozenObject = Object.freeze({})  
  
expect(sealedObject).to.be.sealed  
expect(frozenObject).to.be.sealed  
expect({}).to.not.be.sealed
```

frozen

断言目标对象是冻结的 (无法添加新的属性并且存在的属性不能被删除和修改):

```
var frozenObject = Object.freeze({})  
expect(frozenObject).to.be.frozen  
expect({}).to.not.be.frozen
```

附录 B Vee-Validate 验证规则参考

日期类验证规则

```
date_format:{format}
```

验证的字段必须是指定格式的有效日期。使用任何日期规则时，此规则必须存在。

参数

- **Format:** 日期格式。详细的格式可以参考 `moment.js`。

```
after:{target}
```

用于检测日期型字段值，输入日期必须晚于指定的目标值。

参数

- **target:** 要验证的输入名称。必须具有与 `date_format` 规则相同的格式。

```
before:{target}
```

验证字段必须具有有效日期，并且其值必须早于在目标字段中的日期值。

```
date_between:{min,max}
```

验证的字段必须是指定的两个日期之间的有效日期。

参数

- **min:** 日期的最小允许值。必须采用与 `date_format` 规则相同的格式。
- **max:** 日期的最大允许值。必须采用与 `date_format` 规则相同的格式。

数字类验证规则

```
between:{min},{max}
```

验证字段必须在最小值和最大值限定范围内。

参数

- **min:** 最小值。
- **max:** 最大值。

```
decimal:{decimals?}
```

验证字段必须是数字，并且可以包含指定的小数点数。

- **decimals:** 最大允许小数点数。未通过小数将接受可能包含或可能不包含小数点数字的数字数据。

```
numeric
```

验证的字段只能由数字组成。

```
digits:{length}
```

验证字段必须是数字，并且具有指定的位数。

Length: 数字的位数

字符串验证

```
alpha
```

验证字段只能包含字母字符。

```
alpha_dash
```

验证字段可能包含字母字符、数字、破折号或下划线。

```
alpha_num
```

验证的字段可以包含字母字符或数字。

```
max:{length}
```

验证长度下的字段不能超过指定的长度。

- **length:** 表示最大字符数的数值。

```
url:{domain?}
```

验证的字段必须是有效的网址。可以可选地传递域。支持子域。

- **domain:** 如果网址属于特定网域，请添加另一个检查。tlds 可以提高精度。

```
min:{length}
```

验证长度的字段不应小于指定的长度。

参数

- **length:** 表示最小字符数的数值。

```
confirmed:{target}
```

验证字段必须与确认字段具有完全相同的值。

参数

- target: 目标对照字段的名称。

```
dimensions:{width},{height}
```

添加到验证字段的文件必须是具有指定尺寸的图像 (jpg、svg、jpeg、png、bmp、gif)。

参数

- width: 图片宽度。
- height: 图片高度。

email

验证的字段必须是有效的电子邮件。

ip

验证字段必须有一个有效的 IPv4 值的字符串。

文件类验证规则

```
mimes:[list]
```

添加到验证字段的文件类型应该具有指定的 MIME 类型之一。

参数

- [list] - 逗号分隔的 MIME 类型列表。mimes:image/jpeg|image/png。

可以使用 “” 指定通配符，例如 mimes: image /表示将接受所有图像类型。

```
ext:[extensions]
```

添加的文件在验证下的字段必须指定其中一个扩展名。

参数

- [extensions]以逗号分隔的扩展名列表。例如: ext:jpg、png、bmp、svg。

```
size:{kb}
```

添加到验证字段的文件大小不能超过指定的大小（以千字节为单位）。

参数

- **size**: 最大文件大小（以千字节为单位）。

image

添加的文件验证下的字段必须有一个 image mime 类型 (image/*)。

```
in:[list]
```

验证的字段必须具有指定列表中的值。

参数

- **[list]**以逗号分隔的值列表。例如: in:1,2,3,4。

```
not_in:[list]
```

验证的字段长度不应该在指定值范围内。

参数

以逗号分隔的无效值列表。例如: not_in:1,2,3,4。

```
regex:{pattern}
```

验证的字段必须匹配指定的正则表达式。

- **pattern**: 正则表达式;
- **flags**: 正则表达式标志列表 (可选)。

例如:

```
表达式: ^([0-9]+)$
```

```
// 只接受数值型的数据
```

不应该使用 pipe |。在 HTML 中的正则表达式中，因为它会导致与验证器解析的工作方式发生冲突。解决方法是以编程方式附加字段，并在那里传递正则表达式。

required

验证的字段必须具有非空值。默认情况下，如果验证器具有“空值”，则它们通过验证，除非它们是必需的。那些空值是: empty、string、undefined、null。

好书力荐



本书以 Vue2 的实践应用为根基，从实际示例入手，详细讲解 Vue2 的基础理论应用及高级组件开发，通过简明易懂的实例代码，生动地让读者快速、全方位地掌握 Vue2 的各种入门技巧以及一些在实际项目中的宝贵经验。

本书除了全面、细致地讲述 Vue2 的生态结构、实际编程技巧和一些从实践中得到的经验，还重点介绍如何以组件化编程思想为指导，以前端工程化方法为实现手段来实践 Vue2，通过组件的单元测试和 E2E 测试来保证工程质量。



博文视点Broadview



@博文视点Broadview



责任编辑：陈晓猛
封面设计：王 乐

上架建议：计算机 / 前端技术

ISBN 978-7-121-31068-3



9 787121 310683 >

定价：79.00元